JPL Publication 90-37

p 73

Planning and Reasoning in the JPL Telerobot Testbed

Stephen Peters
David Mittman
Carol Collins
Jacquie O'Meara
Mark Rokey

September 15, 1990

National Aeronautics and
Space Administration

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

JPL Publication 90-37

Planning and Reasoning in the JPL Telerobot Testbed

Stephen Peters
David Mittman
Carol Collins
Jacquie O'Meara
Mark Rokey

September 15, 1990

National Aeronautics and
Space Administration

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

The research described in this publication was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

ABSTRACT

The Telerobot Interactive Planning System is being developed at the Jet Propulsion Laboratory to serve as the highest autonomous-control level of the Telerobot Testbed. This publication describes a recent prototype which integrates an operator interface for supervisory control, a task planner supporting disassembly and re-assembly operations, and a spatial planner for collision-free manipulator motion through the workspace. Each of these components is described in detail. Descriptions of the technical problem, approach, and lessons learned are included.

PAGE INTENTIONALLY BLANK

TABLE OF CONTENTS

TABLE OF FIGURES

1 INTRODUCTION

1.1 PURPOSE AND SCOPE

The Jet Propulsion Laboratory (JPL) is the National Aeronautics and Space Administration's (NASA) lead center for research in space telerobotics; the Goddard Space Flight Center (GSFC) is building the Flight Telerobotic Servicer (FTS), NASA's first operational flight telerobot. The Sequence Automation Research Group (SARG) at JPL is doing research in planning and reasoning for telerobotics and has developed a prototype subsystem and integrated it into the JPL Telerobot Testbed.

The purpose of this publication is to transfer JPL planning and reasoning technology to GSFC, together with insights on how the technology might best be applied. Note that the work is ongoing, and that this publication represents a snapshot of the current state of the research rather than a final report on the technology.

This publication provides a description of Telerobot Interactive Planning System (TIPS) as of the end of FY88, an indication of what to expect in the short term from ongoing research, and observations on parallels with other systems and domains.

Funding for preparation of this publication was provided by the FTS project through GSFC.

1.2 ORGANIZATION

This section describes this publication and refers to external sources of information. Section 2 describes the technical problems targeted by the research. Section 3 describes the technical approach. Section 4 describes the prototype system implemented and demonstrated in FY88. Section 5 describes research in progress as JPL enters FY89, and collects some information about activities going on outside JPL. Section 6 compares the NASA-NBS Standard Reference Model for Telerobot Control System Architecture (NASREM), adopted by GSFC for the FTS, with the architecture of the prototype system described in section 4.

Readers interested in planning the development and evolution of space telerobot systems will find sections 2, 3, and 5 of interest, and may skip the technical details elsewhere. Sections 4 and 6 will be of interest primarily to researchers and those concerned with detailed design and implementation issues.

1.3 REFERENCES

1.3.1 Project Documents

[P1] Telerobotic Technology Program Plan, National Aeronautics and Space Administration, Office of Aeronautics and Space Technology, Revision A, May 1988.

[P2] Telerobotics Project Plan, National Aeronautics and Space Administration, Office of Aeronautics and Space Technology, Information Sciences and Human Factors Division, JPL internal document D-5692, August 30, 1988.

[P3] Functional Requirements for the 1988 Telerobotic Testbed, Jet Propulsion Laboratory, Office of Technology and Applications Programs, Space Automation and Robotics Program, JPL internal document D-3693, Revision 2, May 1988.

[P4] Thread Language Description, Jet Propulsion Laboratory, September 1988.

[P5] NASA-NBS Standard Reference Model for Telerobot Control System Architecture, National Bureau of Standards, Robot Systems Division, Albus, J. S., McCain, H. G., & Lumia, R. Document No. ICG-#001, December 4, 1986.

1.3.2 Project Technical Papers

[T1] C. Collins and M. Rokey, "Planning for the Jet Propulsion Laboratory Telerobotics Project," *Proceedings of the Fourth Annual Artificial Intelligence and Advanced Computer Technology Conference*, Long Beach, California, May 1988.

[T2] D. Mittman, "Audrey: An Interactive Simulation and Spatial Planning Environment for the NASA Telerobot System," *Proceedings of the Fourth Annual Artificial Intelligence and Advanced Computer Technology Conference*, Long Beach, California, May 1988.

[T3] M. Montemerlo and A. deYoung, "The Space Perspective: Man-Machine Redundancy in Remote Manipulator Systems," keynote address to the NATO Advanced Research Workshop on Robots with Redundancy: Design, Sensing and Control, Salo, Lago di Garda, Italy, June 1988.

[T4] S. Peters, "Autonomy Through Interaction: The JPL Telerobot Interactive Planning System," *Proceedings of the SPIE Conference on Space Station Automation*, Cambridge, Massachusetts, November 1988.

[T5] S. Peters, "AI Planner Development," in *Intelligent Robotic Systems: Analysis, Design, and Programming* (S. Tzafestas, Ed.), Marcel Dekker, New York, in press.

[T6] M. Rokey, "Remote Mission Specialist: A Study in Real-time, Adaptive Planning," *IEEE Transactions on Robotics and Automation*, August 1990.

[T7] M. Rokey and S. Grenander, "Planning for Space Telerobotics: The Remote Mission Specialist," *IEEE Expert*, June 1990.

[T8] P. Schenker, R. French, A. Sirota, and J. Matijevic, "The NASA Telerobot Technology Demonstrator," *Proceedings of the SPIE Conference on Space Station Automation*, Cambridge, Massachusetts, December 1986.

1.3.3 General Literature

[G1] E. Cheung and V. Lumelsky, "Motion Planning for Robot Arm Manipulators with Proximity Sensing," *Proceedings of the 1988 IEEE International Conference on Robotics and Automation*, Philadelphia, Pennsylvania, April 24-29, 1988, pages 740-745.

[G2] W. Eggemeyer and A. Bowling, "Deep Space Network Resource Scheduling Approach and Application," *Proceedings of the GSFC Conference on Space Applications of AI and Robotics*, Greenbelt, Maryland, May 1987.

[G3] T. Lozano-Perez, "A Simple Motion-Planning Algorithm for General Robot Manipulators," *IEEE Journal of Robotics and Automation*, Vol. RA-3, No. 3, June 1987, pages 224-238.

[G4] N. Nilsson, *Principles of Artificial Intelligence*, Morgan Kaufmann Publishers, Inc., Los Altos, California, 1980.

[G5] E. Sacerdoti, A Structure for Plans and Behavior, Elsevier Science Publishers, New York, 1977.

[G6] S. Vere, "Planning in Time: Windows and Durations for Activities and Goals," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. PAMI-5, No. 3, May 1983.

An extensive list of additional references is provided in section 5.5 of [P2].

1.3.4 People

Sven Grenander	SARG Supervisor
Stephen Peters	SARG Member, TPR Cognizant Engineer
Carol Collins	SARG Member, CENTeR, Spatial Planning
David Mittman	SARG Member, Audrey, Graphics, Modeling, Interfaces
Jacquie O'Meara	SARG Member, CENTeR, Spatial Planning
Mark Rokey	SARG Member, RMS, Task Planning

1.4 GLOSSARY, ACRONYMS, AND OTHER TERMS

1.4.1 Glossary

Redundant manipulators have more degrees of freedom in their joints than the number of degrees of freedom necessary to specify the position and orientation of the end-effector. Such manipulators have an infinite number of inverse kinematic solutions.

The *Sequence Automation Research Group* is the group at JPL which developed the TIPS system described in this publication.

With *shared control*, the operator, through teleoperation, controls some degrees of freedom of motion while the robot simultaneously controls others.

Thread is the Run-Time Controller (RTC) subsystem development environment and command language. It has a reverse Polish syntax and is similar to the FORTH programming language.

Traded control is the switching between pure teleoperation and pure autonomous operation.

1.4.2 Acronyms

AI	Artificial Intelligence
ART	Automated Reasoning Tool
CAD	Computer-Aided Design
CENTeR	Cluttered Environment Navigator for the Telerobot
CTA	Computer Technology Associates
DMS	Data Management System
DOF	Degree(s) of Freedom
Eos	Earth Observing System
EVA	Extravehicular Activity
FRHC	Force-Reflecting Hand Controller
FTS	Flight Telerobotic Servicer
GSFC	Goddard Space Flight Center
JPL	Jet Propulsion Laboratory
KSC	Kennedy Space Center
MCM	Manipulator and Control Mechanization
MEB	Main Electronics Box

NASA	National Aeronautics and Space Administration
NASREM	NASA-NBS Standard Reference Model for Telerobot Control System Architecture
NBS	National Bureau of Standards
NIP	Network Interface Package
NIST	National Institute of Standards and Technology
NOAH	Networks Of Action Hierarchies
OAST	Office of Aeronautics and Space Technology
OCS	Operator Control Station
ORNL	Oak Ridge National Laboratory
RMS	Remote Mission Specialist
RTC	Run-Time Controller
S&P	Sensing and Perception
SARG	Sequence Automation Research Group
TDRS	Tracking and Data Relay Satellite
TIPS	Telerobot Interactive Planning System
TPR	Task Planning and Reasoning

1.4.3 Other Terms

Audrey	Graphics and Kinematics Simulator
SRI	Developer of the Network Interface Package

1.5 LATE DEVELOPMENTS

This publication is being released several months after the bulk of it was written. Some of the information in this publication is already dated.

Funding of SARG core research was not continued in FY89. Currently, of the activities mentioned in Section 5, only the upgrading of the TIPS internal architecture is funded.

JPL and the Kennedy Space Center (KSC) began a joint project applying TIPS technology to the automation of Space Shuttle Payload Changeout Room operations in FY89.

The National Bureau of Standards (NBS) has been renamed The National Institute of Standards and Technology (NIST). All of the references in this publication refer to the then-current former name.

2 THE PROBLEM STATEMENT

2.1 SPACE TELEROBOTICS

NASA is seeking to augment its space operations capabilities through the introduction of space telerobots, which have both robotic and teleoperator capabilities [T3]. Currently no such telerobot exists. Research is needed to integrate the two capabilities. Research is also needed to enhance robotic capabilities to be useful for one-time-only operations such as satellite servicing.

The first such telerobot to be built will be the FTS. It will have a base site on the Space Shuttle or Space Station, and a remote site at the work-site. For communications, it is to use the Space Station Data Management System (DMS) which introduces a 0.5 second time delay into the base-remote round-trip. Should the base be located on the ground, the time delay through the Tracking and Data Relay Satellite (TDRS) would be up to two seconds. From the point of view of teleoperation, the two delays fall within the same functional range, with equivalent implications for requirements for predictive displays, force feedback, and local automation.

Because a true telerobot is a research product, the FTS is planned to function initially as a teleoperator, and then to evolve, incrementally incorporating enhanced robotic capabilities. The hooks and scars to enable this evolution are to be provided at the outset.

The telerobot is targeted for Space Station assembly and maintenance, and for satellite servicing tasks. All of these involve working with man-made objects in Earth orbit. Models of spacecraft are available, but have proved to contain inaccuracies. For example, a special tool was designed to grapple the Solar Max satellite, but did not fit because of an obstruction not in the satellite plans. Model-based autonomous manipulations can be expected to frequently fail.

Actual telerobot operations in space must be coordinated with and are part of an overall space operations plan. Such plans involve spacecraft resources, crew activities, communication links to the ground, and ground support personnel and equipment. Spacecraft components are often sensitive and have constraints on the thermal, contaminant, and radiation environment to which they may be exposed. These considerations must also be factored into operations plans.

The space environment also places constraints on computing hardware. Specially flight-qualified equipment must be used. This equipment has much slower processing speed and much more limited mass storage capacity than are readily available on ground equipment.

2.2 THE JPL TELEROBOT TESTBED

JPL telerobotics research [P1,P2] is divided into two separately funded categories: core research, and the building of an integrated testbed. Core research is actually a collection of activities addressing the various component technologies needed for telerobotics.

The JPL Telerobot Testbed is a facility for researching and integrating telerobotic technology, demonstrating its feasibility, and developing data to support engineering requirements for flight telerobotic systems [T8].

The research focus of the testbed is on issues which pervade or influence the entire system. Currently these are the implementation of traded and shared control and the handling of time delays from one half to two seconds. *Shared control* refers to the operator controlling, through teleoperation, some of the degrees-of-freedom (DOF) while the robot simultaneously controls

others. *Traded control* refers to switching between pure teleoperation, with the operator at the hand controllers participating in the servo control loop, and pure autonomous operation, where the robot performs manipulations autonomously as commanded by a supervising operator or machine.

Physically, the testbed hosts three six-DOF arms and five video cameras. Two arms are equipped with force-torque sensors and are used for manipulation under position, force, and hybrid control. The third arm supports a stereo pair of cameras. Three additional fixed cameras view the workspace. All five cameras are used both for machine vision and display of the workspace to the operator. Two force-reflecting hand controllers (FRHC) allow the operator to feel forces during teleoperation. A variety of displays is provided.

The JPL Telerobot Testbed system design [P3] consists of six subsystems: five, hierarchically arranged, implement telerobotic technology (figures 1 and 2); a sixth serves as a system health and performance monitor and will not be discussed further here.

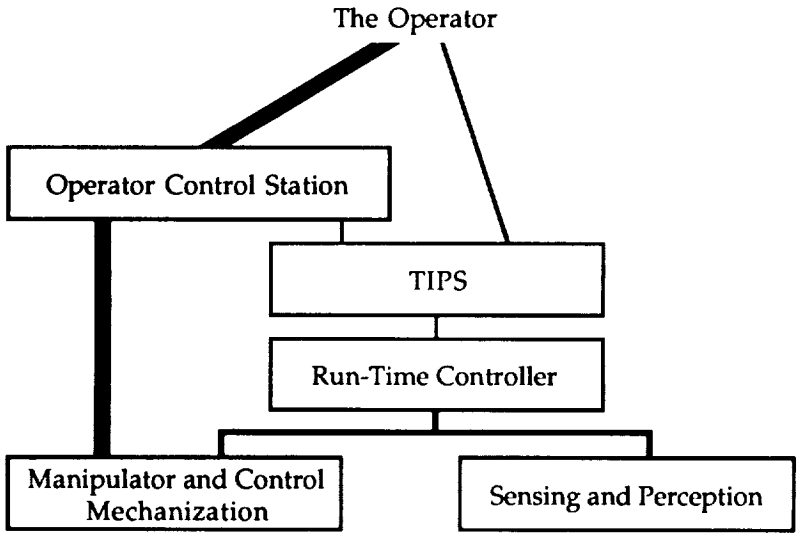


Figure 1. The JPL Telerobot Testbed Control Hierarchy, FY89. Line thickness indicates relative data rates.

The operator sits at the Operator Control Station (OCS), which houses the FRHCs, several monitors including a stereo display and video overlay, and voice input and output devices. Implementation of the OCS was contracted out and is scheduled for delivery to the testbed in FY89.

TIPS is the current implementation of the Task Planning and Reasoning (TPR) subsystem of the testbed. In FY88, TIPS provided automated task planning, gross motion spatial planning, error recovery, and an operator interface to planner knowledge bases. In later years, the TPR subsystem will be expanded to include additional machine reasoning capabilities such as diagnosis. The operator communicates directly with the TPR subsystem terminal located within the OCS.

The Run-Time Controller (RTC) subsystem performs fine motion and grasp planning, plans compliant motions and the application of force, and maintains a geometric database of objects in the workspace.

The Manipulator and Control Mechanization (MCM) subsystem implements control laws. Trajectories may be in either task or robot configuration space. Force and hybrid force-position control are also supported. MCM also provides built-in macros capable of autonomously performing local tasks such as tightening a bolt or part mating.

The Sensing and Perception (S&P) subsystem implements machine vision. Customized pipeline processors are used. The capabilities are tracking the motion of a rotating known object and verification of the position of a non-moving known object. Both functions determine the position and orientation of the objects.

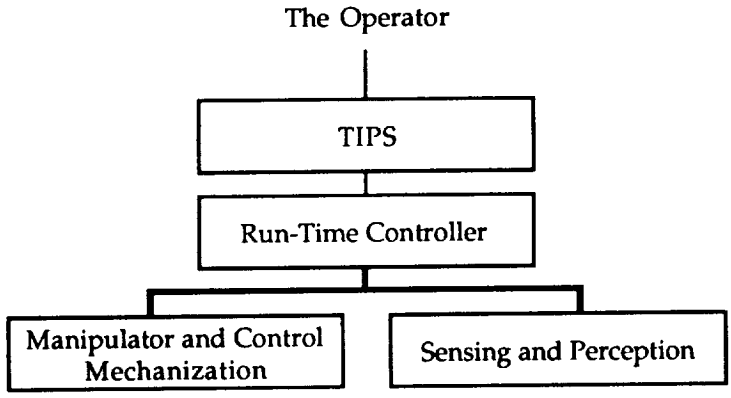


Figure 2. The JPL Telerobot Testbed Control Hierarchy, FY88.

2.3 PLANNING AND REASONING

Intelligence in the testbed is distributed across the testbed subsystems. Vision processing takes place wholly within S&P. Robot dynamics issues are handled by MCM. Precise geometric modeling and monitoring of manipulability are done by RTC. Reasoning on the over-all task is done by TPR. TPR knowledge and reasoning capability is not a super-set of the capabilities in other subsystems, but instead complements them.

The goal for the TPR autonomous capability is to accept one or more high-level servicing instructions from the operator, such as replacement of a module, and to command the RTC to execute primitive operations which will result in accomplishing the over-all task. The primitives include both manipulation and sensing commands. Feedback is returned by RTC upon completion and, in the case of failure, TPR must modify its course of action to overcome the difficulty.

Just as the operator can, through teleoperation, assist where autonomous MCM and RTC capabilities fail, the operator must also fill in for shortcomings in S&P and TPR capabilities. An operator interface to high-level machine reasoning must be provided.

Satellite servicing involves disassembly and re-assembly necessary to access parts. Fasteners must be detached and reattached. Tools must be employed. Plans must include teleoperation of

difficult manipulations such as the handling of flexible thermal blankets. Telerobot capabilities to be employed are also a part of servicing plans.

Spatial paths must be found for moving objects and manipulators through the workspace. The telerobot itself must be located where it can achieve the necessary manipulations.

In general, the management of each specific manipulation or sensing operation is handled by the subsystem involved. The greatest knowledge of detailed manipulation and sensing strategies is local to the appropriate subsystem. Success of such subsystem autonomous capabilities depends upon their being applied in the appropriate task context. Such context management, and related diagnostic reasoning, must also be done by a higher-level planning subsystem.

Telerobot activities, as they are a part of space operations, must fit into overall spacecraft operations plans. Such plans include diverse activities and allocation of limited resources such as power and communications links.

3 APPROACH

3.1 TECHNOLOGY OUTLOOK

3.1.1 State of the Art

Telerobotics research treads new ground. The JPL Telerobot Testbed is currently the main facility pushing some of the technologies required for telerobotics and may be expected to continue in this role.

Robotics still focuses on the automation of repeated processes. The telerobotics objective of supporting one-time-only robotic activity in one-time-only worlds will continue to be extremely challenging.

The greatest successes in automation are in controlled worlds — either the idealized worlds of machine processes or in controlled real-world environments. Telerobotics objectives place research goals outside of these more comfortable bounds. Goal-driven systems where desired predicates are states in the real and messy external world will be forced to rely on sources of knowledge external to themselves.

Artificial intelligence (AI) theory and application are quite far apart. Practical solutions involve utilizing assumptions allowed by the specific problem space; performance of general purpose reasoning paradigms is generally slow. As many problems are NP-complete, hardware and parallel architectures cannot be expected to have a major impact on basic technology. Current interest may, however, spark useful parallel formulations.

3.1.2 State of the Testbed

Limitations in sensing will force use of a model-driven approach to autonomy and dependence upon the operator for observation. Currently, only force-torque sensors, joint encoders, and machine vision cameras are present. These are used for determination of the position of known modeled objects, one at a time. Visual servoing has been demonstrated stand-alone, and may be integrated into the system in the near future. Addition of tactile sensing, laser scanning, structured light, and proximity sensing have been suggested. These could support building three-dimensional maps of unknown environments and active collision avoidance.

The testbed computing architecture is beginning to evolve and is expected to come to support a greater number of concurrent activities directed in a less constrained control structure. Currently, the testbed subsystems perform much like subroutines in that they are called by other subsystems higher in the strict tree hierarchy. An architecture evolution will be necessary to provide greater concurrency and overall system robustness, and can be expected to take place over the course of several years. Already, early system designs are being revised: there has been a move from microVAX's to Sun Workstations and a move of Ethernet Network Interface Package (NIP) interfaces to VME busses.

Development will always be bottom-up: MCM capabilities will become mature before RTC capabilities, which will in turn mature before TPR capabilities.

As the testbed is a large facility requiring significant resources, high-risk research will not be performed on it. The level of robotics technology incorporated in the testbed will always be

behind the state of the art — except in the area of specifically telerobotic technologies. These include traded and shared control, and one-time-only robotics.

The integrated telerobot capabilities will evolve (figure 3) to address more and more of an overall problem, beginning earlier and earlier in the life-cycle of the problem. At first, only the required manipulations will be performed; later, machine reasoning will be able to figure out what is wrong from symptoms of off-nominal behavior.

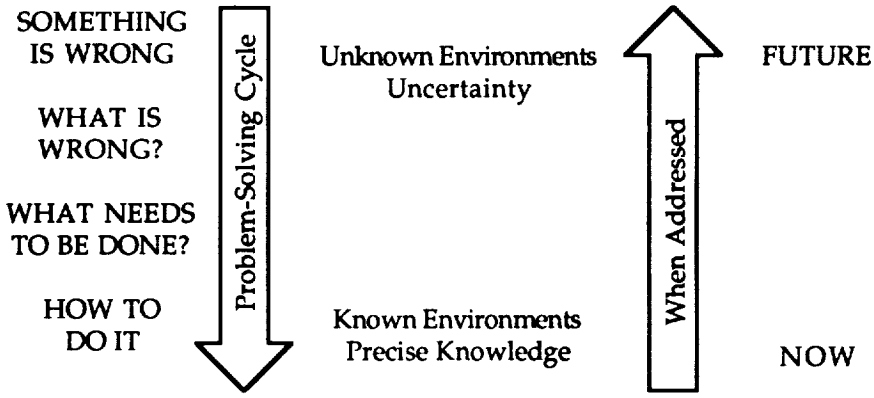


Figure 3. Issues and when they are addressed.

3.2 STRATEGY

3.2.1 Applications Research

As an applications research group, SARG tries to attack real research issues, focussing on practical results in real-world domains.

Our primary technology goals for the planner were:

- Speed
- Error Recovery
- Spatial Planning
- Operator Interface

All of these are open research issues. Previous planners like SARG's own DEVISER [G6] which use logical chaining and search to find a solution become unusably slow when applied to realistically complex problems. Robotic interaction with an external environment requires the ability to recover from errors if any degree of usefulness is to be achieved. Spatial planning is an open NP-complete problem for real-world robotic manipulators applied to non-repetitive tasks. Access to and utilization of external knowledge from an operator is necessary to overcome the shortcomings in planning & reasoning and sensing & perception technology in the near term.

In order to be practical, whole systems or subsystems must be developed. One can neither wait for technology nor assume one's yet undemonstrated research product, currently in development, will work. Therefore, interim stop-gap components must be added to complete the necessary functionality. Enhancements are made whenever technology is ripe.

3.2.2 Rapid Prototyping

In order to be sure to always work on the right problems, properly posed, we try to build a system and run it on a real problem as soon as possible. The importance of this approach cannot be stressed enough.

One example of a benefit from this approach is the identification of the importance of the role of the operator as a knowledge source for automated operations. Another was the allocation of gross motion and fine motion spatial planning responsibilities between RTC and TPR (see section 4.1), which was determined during the process of developing and driving a graphics simulation of the satellite servicing scenario.

3.2.3 Target Functional Design

In the fall of 1987, a very rough functional design document was written. It included task domain description, world modeling suggestions, description of operator interface capabilities, and it defined target task planner robustness. Many of the capabilities described became part of the system described in this publication, some did not, and some features and capabilities beyond those mentioned were implemented.

The overall strategy for the operator interface was to first provide for examining and updating any portion of the knowledge within the planner through primitive transactions — a kind of knowledge debugging capability. Later, cognitive interfaces would be developed from these primitives.

The task planner was to handle satellite disassembly-assembly operations and accept multiple goals along the same or unrelated assembly paths. It was to handle multiple tools, loss of tools, and manipulation failures. One concept, retained from experience with a prior prototype planner, was to plan task-space activity and robot activity independently.

It is important that this was an internal document. Internal activity allowed the freedom to do creative work, while providing direction as needed.

3.2.4 Applicable Concepts and Suggestions

Since teleoperation is a world state wild-card, develop an interface to the world model which would allow the operator to teleoperate the model into agreement with the task-space.

A simulation is an ideal world. One can therefore always develop a planner which will drive a simulator as desired. On the other hand, the real world is messy and unpredictable. In order to test systems which will eventually interact with the real world, the testing loop must also pass through the external world. This can be accomplished by putting an operator in the feedback loop during testing.

Machine representations and reasoning algorithms should be compatible with the way the operator naturally thinks. If not, the operator will be unable to assist the machine, even when the machine has almost found the solution to a problem. Machine reasoning should accept constraints from the operator without having to understand them.

To make it easier to engage autonomous operations, perform teleoperation as part of a plan known to the machine. This provides context which makes it much easier for the operator to

tell the machine what happened during teleoperation. Step completion plus off-nominal results would suffice.

Expect reasoning engines to fail, to make mistakes and wrong choices from time to time. Such errors, as they are inevitable, cannot be allowed to force the remainder of the mission to be completed via teleoperation.

The best knowledge of robotics will be in the MCM and RTC subsystems. These subsystems will locally attempt recovery before they return with a manipulation failure. The assistance TPR will provide will be to adjust the context in which MCM and RTC make their attempts.

Use least-commitment planning strategies and try to minimize backtracking. Otherwise planning speed will be slow and error recovery will be difficult.

To the extent possible, technology developed should not be tied to specific telerobot hardware. It should, however, be proven on current testbed configurations.

3.3 PRIORITIES

For differing reasons, the following parts of the overall problem were left for later years:

- Positioning of the Robot Relative to the Workspace
- Handling of Time-Related Constraints
- Integration into the Overall Space Operations Plans
- Reasoning About Force
- Subdivision of Single Operations
- Reasoning From First Principles
- Intentional Sensing
- Diagnosis

Top priority for research was development of a task planner capable of planning and executing servicing tasks involving disassembly and re-assembly, and incorporating error recovery. Execution primitives consisted of a set of commands concurrently defined and developed by RTC.

Useable gross motion spatial planning algorithms did not exist. And they are needed for all robotic manipulation. As this is a necessary and missing technology, and as this is an open research issue, it was given high priority for core research work.

Recognition of limitations in sensing available in the testbed, the likelihood of failures in execution due to unmodeled complexity in the task-space, and the need to support traded control placed development of an operator interface as a necessary component of the initial system.

As RTC development was concurrent with development of other subsystems, a simulator was needed to test the task planner. Note that unlike traditional knowledge-based system development, the knowledge to be modeled did not exist beforehand.

In keeping with the strategy of building a complete system as soon as possible, considerable effort was placed on subsystem design and integration. As the gross motion spatial planner was a high-risk development, a simple simulator of it was developed. Aside from the gross motion spatial planner, all of the components mentioned were considered necessary to a minimal functional capability. Funding constraints would limit the robustness and sophistication of the operator interface and the task planner; augmented funding would allow some of the postponed issues to be addressed.

4 THE TELEROBOT INTERACTIVE PLANNING SYSTEM

4.1 INTERFACES AND ROLE WITHIN THE SYSTEM

4.1.1 External Interfaces

The TPR subsystem has external interfaces with the operator, the OCS subsystem and the RTC subsystem.

The FY88 TIPS implementation included a subset of the operator interfaces planned for the system. Outputs to the operator include: a wire-frame perspective graphics display of all objects in the workspace; a display of the object-position-inheritance tree; graphics displays of the joint angles, joint stops, and singularities of all three manipulators; and a trace of the messages passed between the task planner and the spatial planner. Controls provided include: a means for updating the position and orientation of objects, updating manipulator joint angles, and adjusting the point of view of real and virtual cameras; the ability to make objects visible and invisible; commands to initiate the satellite servicing scenario; and commands to select RTC simulation or execution.

OCS interfaces will not be implemented until FY89 and are not part of the current version of TIPS.

The RTC interface consists of commands in RTC's Thread language [P4]. The only Thread commands issued by TIPS at execution time were simple manipulator move commands. Thread was also used for the off-line portion of spatial planner processing. (Note that Thread definition and development were proceeding concurrently.)

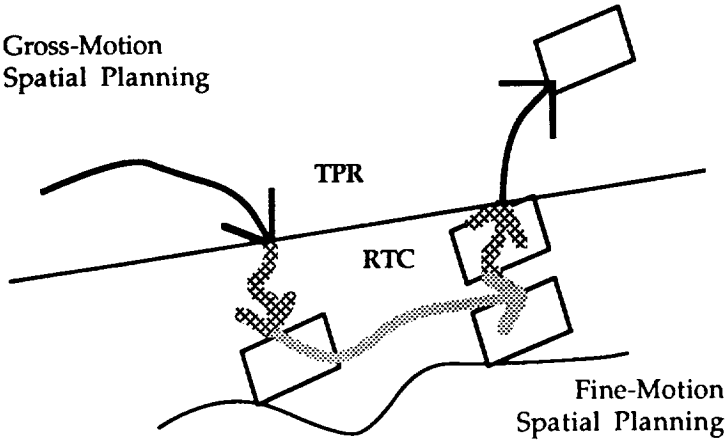


Figure 4. Allocation of spatial planning responsibilities.

A major portion of the RTC-TPR functional interface is the division of spatial planning responsibilities (figure 4). Gross motion spatial planning is performed by TPR and fine motion spatial planning is performed by RTC. The gross motion spatial planner must find a path for the manipulator and carried object through the workspace without contacting any other object. Great precision is not necessary as long as the manipulator is far enough away from obstacles. On the other hand, the fine motion spatial planner is concerned with precise interactions between objects: controlled collisions, part mating and de-mating, and compliant manipulations.

One qualitative distinction between the spatial planning domains is that fine motion planning is concerned with planning end-effector motion while gross motion planning is concerned with whole-arm motion.

4.1.2 TIPS Internal Organization

TIPS [T5, T4] consists of three major software modules (figure 5): the Remote Mission Specialist (RMS) task-planner [T1,T7,T6], the Cluttered Environment Navigator for the Telerobot (CENTeR) gross-motion spatial planner [T1], and the Audrey simulator [T2].

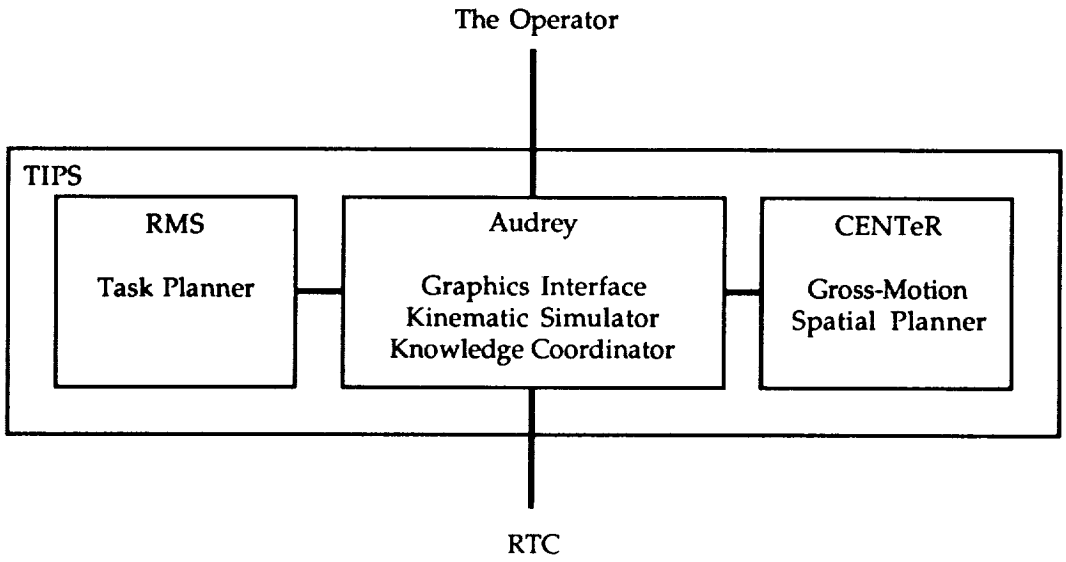


Figure 5. TIPS major modules.

RMS works with semantic knowledge of the robot and workspace. CENTeR works with geometric knowledge of the robot and workspace. Audrey maintains a geometric model of the workspace and kinematic models of the manipulators, and it coordinates messages passing between the operator, RMS, CENTeR, and RTC. When RMS issues a command for free motion of an arm, it is expressed as something like: "move the left arm end-effector to the approach point for the screwdriver." Audrey determines the set of manipulator configurations currently implied by the position of the screwdriver in the RMS request, then passes these alternative goals to CENTeR as a path-find request. Audrey receives the path returned from CENTeR and formats it into RTC command syntax, or simulates the motion itself (figure 6).

TIPS modules differ not only in problems addressed and knowledge domain, but also in general software construction: RMS is a rule-based expert system written in the Automated Reasoning Tool (ART) language; CENTeR is a collection of functions written in Ada and Common LISP; and Audrey makes heavy use of object-oriented Symbolics Flavors in addition to functions in Common LISP. This is the reason for the difference in the structure of the descriptions presented below. It is intended that these descriptions provide enough detail to serve as a guide to one reading the software source code.

4.1.3 Demonstration Task Scenario

The Telerobot Testbed project had originally developed a target servicing task scenario based upon the Solar Maximum Repair Mission. This scenario was augmented with additional parts and disassembly paths to provide a richer problem space for the TIPS task planner.

The satellite model contains a Main Electronics Box (MEB), attached with four screws to the satellite chassis. Two electrical connectors also connect the MEB to the satellite. A bracket, held in by two screws, inhibits removal of the MEB. All this is behind a cover which is also attached by four screws to the satellite. Additional parts and replaceable units were also defined to ensure robustness of successful task planning algorithms, but were not present in the final system as demonstrated.

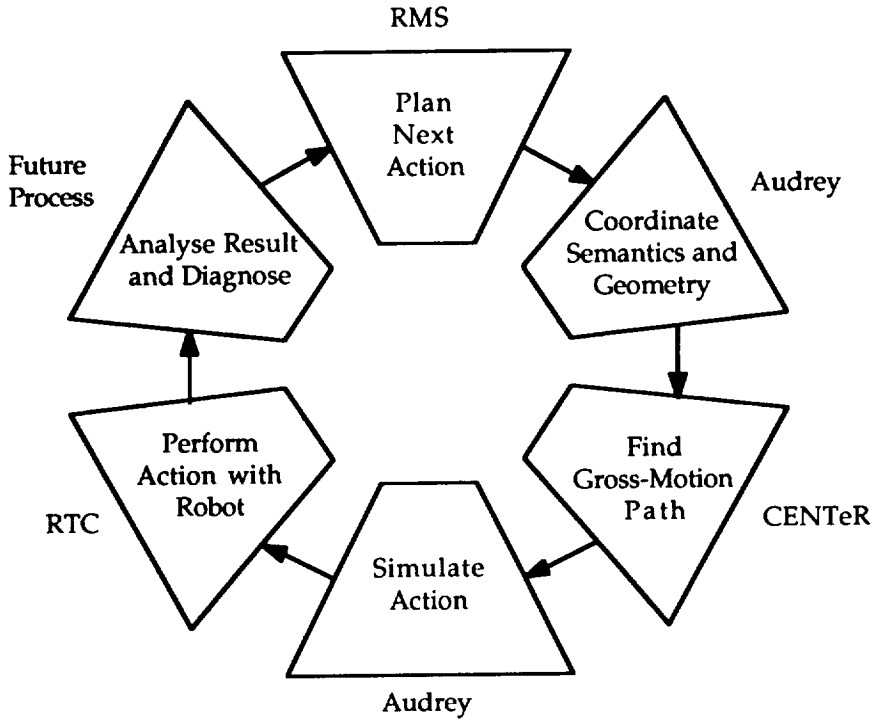


Figure 6. The TIPS planning and control message loop. Opportunities for operator interaction exist between each process in the message loop.

4.2 THE TASK PLANNER

4.2.1 Planner Architecture

RMS is made up of two stages of plan generation. Stage One is responsible for converting high-level directives into a series of commands which tell what specifically needs to be done in the task-space. This series of commands is then used as input to the second major stage of the planner, where the commands are converted into primitives which are executable by the Run-Time Controller subsystem. While Stage One tells what to do in the task-space, Stage Two decides how to do the operations by examining each step in the plan and adding knowledge of

the robot, its resources, and the current state of the world onto the structure to derive the final plan output.

The reasoning behind this organization is that a high-level controller should provide some type of intelligent or resilient approach to error recovery while operating remotely in the task environment. For a space based robotic operation, we know the general set of things which need to be done to accomplish a given task. However, since the task-space cannot be tweaked to factory tolerances (a few millimeters make a great deal of difference in robotic terms), the actual state of the task-space is unknown until the robotic vehicle begins its operations. This design generates an overall blueprint, and then as the robot craft is in the process of servicing, expands the low level details as it goes, dynamically.

This first working version of RMS makes several simplifying assumptions. First, we assumed that RTC commands were atomic in nature. That means that, for instance, an open the door command would be one command provided by RTC, not a collection of moves, grasps, and pivots. We assumed that all servicing operations can be done by a single arm, and so support no dual arm manipulations currently. Finally, we assumed that there are no conditional plan elements in the output of Stage One. This means the partially ordered graph created by Stage One is made up only of AND branches, and every node in the graph must be achieved in order for the servicing operation to be complete.

4.2.1.1 Stage One

The planning paradigm used for Stage One is a derivative of the Networks Of Action Hierarchies (NOAH) system [G5]. A goal is defined to be an action which defines some state. Goals are given to Stage One in the form of *action-units*, frames containing an action and associated parameters, such as the object to which the action is applied. The output of Stage One is a network of action-units, where each action is at a low enough level of abstraction to be understood by Stage Two.

The plan generation process starts with an initial goal (or net of goals) which describe the entire operation to be performed. Each net node whose action is not yet understandable by Stage Two is macro-expanded into a lower level net of actions, defined in the planner's knowledge base. As the number of levels increases, the overall level of abstraction of the plan decreases, until finally all the nodes in the network are understandable by Stage Two. This structure can be generated very quickly, and allows input goals to be specified at varying levels of abstraction.

Consistency of plans, as expanded from one level to the next, is maintained by specialized pieces of code called critics. Critics examine a newly generated level, looking for inconsistencies. When one is found, a critic dealing with that type of inconsistency takes action by reordering the plan network, eliminating repeated nodes, or removing nodes whose action has already been taken in the plan space.

Though we have made some customizations, Stage One generally follows the NOAH paradigm. This paradigm was selected because of the speed of generation of plans it provides and the flexibility in the types of plans it can build. It is important to note that any number of paradigms could have been chosen, without impact to the overall planner, speaking from an architectural standpoint. Stage One is not restricted to using NOAH, but NOAH is an excellent tool for the types of problem spaces on which this research is focused.

Stage One deals only with the task-space. No robotic information is included. The level of command it outputs is roughly equivalent to a set of assembly instructions, and so its output could also be used to guide a teleoperator with a remote robot vehicle, or to guide a manned Extravehicular Activity (EVA) operation. Stage One was designed to be a static, non-interactive process for producing overall work plans for review by a teleoperator. It may be noted, however, that NOAH provides convenient break-points in the planning process, where a user could guide Stage One, should such guidance be needed.

4.2.1.2 Stage Two

The input to Stage Two is the bottom-level network built by Stage One, or a similar net supplied by the user. The output of Stage Two is in the form of commands which can be executed by the RTC subsystem. While Stage One can be thought of as macro-expanding one level of a plan to another, less abstract level, Stage Two is actually a net-walker, starting at the net beginning and moving through it, commanding the robot as it goes.

The method by which this command generation is done is essentially a nested looping structure, where the data items flowing through this structure are individual action-units, described above. Net nodes that have not been acted out are marked. From those marked, one is chosen to be worked on by a set of scheduling heuristics which make the best choice of next node and designate the robot resources to be assigned to act on it.

Once an action-unit is chosen for expansion and the necessary robot resources have been selected for it, the action-unit flows through a set of rules which determine what commands need to be sent to carry out the specified action. This set of rules is made up of two types, *expanders* and *completers*. Expander rules look at the state of the world and select the appropriate command to be sent. Once selected, the command is dispatched to be executed. On execution, a status message, which is caught by a completer rule, is returned to the planner. A completer is a specialized piece of code which accepts an incoming message and makes appropriate updates to the knowledge base. Notice that no knowledge base updating is done by the expanders. Updates are made strictly through the completers, based on feedback information. Expanders and completers work in tandem until the intent of the action-unit has been achieved. The entire process then repeats for the remainder of the unachieved action-units.

The expander-completer mechanism allows for on-the-fly plan adaptation. The same structure that accepts a positive response and updates for the next command can accept failure messages and update so the recovery from these failures is done as a part of the normal planning process.

Stage Two, using the expander-completer mechanism, provides dynamic, reactive generation of commands, as opposed to the static generation of Stage One. Stage Two does not generate the entire sequence before it begins execution. Rather it uses the assembly instructions provided by Stage One and does a run-time expansion of those instructions, a step at a time, based on the feedback of the world in which it works. Stage Two deals with robot resources and figures out how to do the what to do Stage One has directed. Stage Two is also extremely fast, and is able to generate commands in real-time, actually faster than the robot can execute them.

4.2.2 Knowledge Base Description

The knowledge base used for RMS provides semantic descriptions of the objects in the world. These objects are derived from a set of high-level types, which cover generic descriptions for

the parts of the task-space, the robot arms and any tools used. A list of these high-level types is provided below.

Type:	slot
Feature-of:	<parent-object>
Contains:	<corresponding replaceable-unit objects>, nothing
Status:	filled, empty

Type:	grasping-appendage
Feature-of:	<parent-object>
Status:	gripped, ungripped

Type:	moveable-fixed-unit
Permanently-attached-to:	<parent-object>
Semantic-location:	closed-position, opened-position
Status:	open, closed

Type:	replaceable-unit
Seated-in:	<corresponding SLOT object>
Fastened-by:	<fasteners>
Stowed-in:	toolbox
Carried-by:	one-arm, two-arms
Status:	seated-and-fastened, seated-and-unfastened, stowed, carried

Type:	fastener
Permanently-attached-to:	<parent-object>
Status:	engaged, disengaged

Type:	tool
Stowed-in:	toolbox
Carried-by:	<one-arm, two-arms>
Status:	stowed, held

Type:	arm
Semantic-location:	<arm-position>
Holds:	<some object>, nothing
Status:	engaged, unengaged

Type:	moveable-replaceable-unit
Seated-in:	<corresponding SLOT object>
Fastened-by:	<fasteners>
Stowed-in:	toolbox
Carried-by:	one-arm, two-arms
Status:	seated-and-engaged, seated-and-disengaged, stowed, carried

From these types, subtypes are defined, which cover specific kinds of objects in the task-space. A partial list is given below.

Subtype:	Door
Type:	moveable-fixed-unit
Permanently-attached-to:	satellite-chassis
Semantic-location:	closed-position, opened-position
Status:	open, closed

Subtype:	MEB
Type:	replaceable-unit
Seated-in:	MEB-slot
Fastened-by:	<MEB-screws>
Stowed-in:	toolbox
Carried-by:	one-arm
Status:	seated-and-fastened, seated-and-unfastened, stowed, carried

Subtype:	wrench
Type:	tool
Stowed-in:	toolbox
Carried-by:	one-arm
Status:	stowed, held

Subtype:	arm
Type:	arm
Semantic-location:	<arm-position>
Holds:	<some object>, nothing
Status:	engaged, unengaged

Subtype:	MEB-screw
Type:	fastener
Permanently-attached-to:	MEB
Status:	engaged, disengaged

Finally, from these subtypes, *instances* are created, which are the specific data elements manipulated by RMS. In the example below, MEB-1 is an instance of MEB, which is a subtype of *replaceable-unit*.

```
(defschema MEB-1
   (type replaceable-unit)
   (instance-of MEB)
   (Seated-in MEB-slot-1)
   (Fastened-by MEB-screw-1 MEB-screw-2 MEB-screw-3 MEB-screw-4)
   (Stowed-in MEB-tool-box-slot)
   (Carried-by one-arm)
   (status seated-and-fastened))
```

The reason for representing objects in this manner is that high-level operators can be defined which are capable of handling not only multiple instances of objects, but entire collections of related objects. These type definitions provided the basis for defining the actions described in the action-unit data structure and are referenced in both Stage One and Stage Two of RMS.

4.2.3 Stage One Description

The job of Stage One is to take a task described as an action-unit and re-describe it in terms of a collection of less abstract action-units. In the example below, an action-unit to swap units of type *replaceable-unit* is expanded into four equivalent action-units which describe the task at a less abstract level. Notice that in these less abstract descriptions, the associated action is one which is still defined to operate on the *replaceable-unit* type.

```
(defrule expand-swap-replaceable-units
   (active-rule-set level-1)
   (current-expansion-level ?l)
   (schema ?action-unit
         (expansion-level ?l)
         (action swap-replaceable-units)
         (object ?object)
         (toolbox-replacement ?replacement))
=>
   (bind ?new-l (incf ?l))
   (bind ?name-1 (getname))
   (bind ?name-2 (getname))
   (bind ?name-3 (getname))
   (bind ?name-4 (getname))
   (assert
         (schema ?name-1
               (action extricate-replaceable-unit)
               (object ?object)
               (expansion-of ?action-unit)
               (expansion-level ?new-l))
         (schema ?name-2
               (action store-in-toolbox)
               (object ?object)
               (expansion-of ?action-unit)
               (expansion-level ?new-l)
               (comes-after ?name-1))
         (schema ?name-3
               (action get-from-toolbox)
               (object ?replacement)
               (expansion-of ?action-unit)
               (expansion-level ?new-l)
               (comes-after ?name-2))
         (schema ?name-4
               (action captivate-replaceable-unit)
               (object ?replacement)
               (expansion-of ?action-unit)
               (expansion-level ?new-l)
               (comes-after ?name-3))))
```

This example is typical of the primary set of rules which operate in Stage One. To aid this process of redefining high-level actions into lower level ones, Stage One also has a set of supporting rules which describe constraints, to insure correct expansion of high-level action-units. For example, the high-level action to get obstructions out of the way of a particular component is supported by a rule which tells it that in this task-space, under certain conditions, a certain door is an obstruction for the fasteners of the component. It is through this set of constraint rules that interaction of task-space objects is modeled.

```
(defrule door-obstructs-MEB-screws
   (goal (obstructs ?door ?MEB-screw ?level))
   (active-rule-set level-1)
   (current-expansion-level ?level)
   (schema ?MEB-screw
         (instance-of MEB-screw)
         (permanently-attached-to ?MEB))
   (schema ?MEB
         (instance-of MEB)
         (seated-in MEB-slot-1)
         (status seated))
   (schema ?door
         (instance-of door)
         (status closed))
=>
   (assert (obstructs ?door ?MEB-screw ?level)))
```

The reason we use complicated-looking rules to describe what objects obstruct others (rather than a look-up table, etc.) is that often whether or not one part obstructs another depends on certain state parameters in the world. In the example above, the door only obstructs the fasteners if it is closed. Should the door be open already, it is not an obstruction and does not need to be gotten out of the way.

Finally, to maintain plan consistency there are certain rules which are called critics, which look for flaws in the plan and repair them. For the task-space described in this publication, we only needed to implement one critic, which looked for duplicate action-units being generated and simply eliminated any extras. This duplication occurs when objects are being marked as obstructions. Often one object is an obstruction to multiple items of interest, resulting in the same operation being commanded more than once. When this situation occurs, the eliminate-redundant-action-units critic examines the plan and cuts out any duplications.

4.2.4 Stage Two Description

Stage Two takes as input the partially ordered set of action-units generated by Stage One (or input by an operator) and produces commands which can be executed by the RTC. It generates the command dynamically, in an opportunistic fashion, in an effort to provide some kind of localized error recovery. The flow of control in Stage Two conceptually follows the pseudo-code given below, with action-units as the data items flowing through this control path. (This control path is given only to aid in visualizing the control flow. The actual code is written in a rule based language, and does not resemble this.)

```
while there is an unfinished action-unit do
   mark current expansion candidates;
   select one with scheduling heuristics;
   repeat
         select and fire expander;
         dispatch command;
         receive feedback message;
         select and fire completer;
   until action-unit is accomplished or in need of re-scheduling
endwhile
```

Because we are using a least-commitment strategy to order the tasks at Stage One, at execution time there is need to choose which of a number of logically parallel tasks to work on. To do this, we incorporated a set of scheduling heuristics, which look at all available action-units to be worked on and choose one, based on programmer-defined criteria. An example scheduling heuristic follows, together with a list of those incorporated in the demo version of RMS.

```
(defrule use-tool-being-held-heuristic
   (declare (salience -111))
   (active-rule-set level-2)
   (schema ?action-unit
         (exec-status ELIGIBLE)
         (action ?action)
         (object ?object))
   (schema ?object
         (instance-of ?object-class))
   (schema ?arm
         (type arm)
         (holds ?tool))
   (schema ?tool
         (instance-of ?tool-class))
   (?action ?skill ?object-class ?tool-class)
   (approach-point ?skill ?object ?arm-position ?other-position)
   (NOT (schema ?action-unit
               (forbidden-arm ?arm)))
=>
   (modify
         (schema ?action-unit
               (exec-status READYED)
               (arm ?arm)
               (approach ?arm-position)
               (de-approach ?other-position)
               (tool ?tool)
               (skill ?skill))))
```

complete-object-in-transit-operation-heuristic
use-stowed-tool-if-forbidden-arm-holds-duplicate-heuristic
pick-uncluttered-arm-when-no-tool-required-heuristic

pick-uncluttered-arm-when-tool-required-heuristic
pick-at-random-when-tool-required-heuristic
pick-at-random-when-no-tool-required-heuristic

To get a better idea of what an expander and a completer are, think of commands represented in terms of preconditions and post-conditions, as is done in means-end analysis. An expander (see example below) is made up of the corresponding preconditions part, which describes the physical world semantically. It also has an intent portion which is described by the action-unit with which it is associated. The action-unit provides direction, so the expander will know when to come active. This eliminates the need for search and backtracking, as an expander only becomes active when the class of action-units to which it applies is currently active, and when the conditions of the world are correct.

```
(defrule ATTACH-THREADED-FASTENER-expander
   (active-rule-set level-2)
   (schema ?action-unit
         (exec-status READYED)
         (action ATTACH-THREADED-FASTENER)    ;Intention Part
         (object ?object)                ;Fire only when you are
         (tool ?tool)                    ;expanding an attach-
         (arm ?arm)                      ;threaded-fastener on a
         (skill ?skill)                  ;given object…
         (approach ?approach))
   (schema ?tool                                ;World State Part
         (status held))                  ;…AND when the world
   (schema ?arm                          ;state parameters are correct
         (holds ?tool)                   ;for its execution.
         (semantic-location ?approach))
=>
   (assert (command-ready ATTACH-THREADED-FASTENER
         ?skill ?object with ?tool with ?arm)))
```

The left-hand side of a completer is composed of a message pattern which is returned by the robot, indicating either success or failure, and the reason for the failure. The right-hand side is the set of updates made to the knowledge base, and corresponds to the post-conditions part of a means-ends analysis rule. Updates are also made to the action-units at this time.

```
(defrule ATTACH-THREADED-FASTENER-success-completer
   "Command was executed successfully."
   (active-rule-set level-2)
   (schema ?action-unit
         (exec-status READYED)
         (de-approach ?de-approach))
?r <-    (command-executed ATTACH-THREADED-FASTENER  ;message portion
               ?skill ?object with ?tool with ?arm)
=>
   (retract ?r)
   (modify
```

```
            (schema ?arm
                  (semantic-location ?de-approach))      ;update portion
            (schema ?action-unit
                  (exec-status COMPLETED))))
```

In this example, the completer looks for a *command-executed* message and makes the appropriate updates. In the example which follows, the completer looks for a command-failed-case-1 message which means the arm was kinematically unable to perform the desired operation, so mark that action-unit as unable to be solved with the specific arm, and try again.

```
(defrule ATTACH-THREADED-FASTENER-failure-1-completer
   "Failure due to case 1, a fatal arm kinematic violation."
   (active-rule-set level-2)
?r <- (command-failed-case-1 ATTACH-THREADED-FASTENER ?skill ?object
         with ?tool with ?arm)
         (schema ?action-unit
               (action ATTACH-THREADED-FASTENER)
               (object ?object)
?r1 <-         (exec-status READYED)
?r2 <-         (arm ?arm)
?r3 <-         (skill ?skill)
?r4 <-         (approach ?approach)
?r5 <-         (de-approach ?de-approach)
?r6 <-         (tool ?tool))
=>
   (retract ?r ?r1 ?r2 ?r3 ?r4 ?r5 ?r6)
   (modify
         (schema ?action-unit
               (forbidden-arm ?arm))))
```

In the event of a failure, a corresponding failure completer makes a different update, which in effect means that the planner is able to plan as it goes, checking neither unnecessary hypotheses for which option to perform at a given time nor varied resultant states of execution. It makes its decisions as it goes, opportunistically, following the road-map produced by Stage One and correcting for localized failures along the way. A list of expanders and completers implemented in the demonstration version of RMS is given below.

DETACH-THREADED-FASTENER-expander
DETACH-THREADED-FASTENER-completer
OPEN-MOVEABLE-FIXED-UNIT-expander
OPEN-MOVEABLE-FIXED-UNIT-completer
CLOSE-MOVEABLE-FIXED-UNIT-expander
CLOSE-MOVEABLE-FIXED-UNIT-completer
INSTALL-REPLACEABLE-UNIT-expander
INSTALL-REPLACEABLE-UNIT-completer
REMOVE-REPLACEABLE-UNIT-expander
REMOVE-REPLACEABLE-UNIT-completer
GET-FROM-TOOLBOX-expander

GET-FROM-TOOLBOX-completer
STORE-IN-TOOLBOX-expander
STORE-IN-TOOLBOX-completer
MOVE-MANIPULATOR-to-position-for-skill-when-tool-required-expander
MOVE-MANIPULATOR-completer
MOVE-MANIPULATOR-to-position-for-skill-when-tool-not-required-expander
MOVE-MANIPULATOR-to-position-for-tool-acquisition-expander
GET-TOOL-expander
GET-TOOL-completer
MOVE-MANIPULATOR-to-position-for-tool-deacquisition-expander
REPLACE-TOOL-expander
MOVE-MANIPULATOR-for-tool-deacquisition-for-opposite-arm-expander
REPLACE-TOOL-for-use-by-opposite-arm-expander
REPLACE-TOOL-completer
MOVE-MANIPULATOR-to-unobstruct-the-workspace-expander
MOVE-MANIPULATOR-without-tool-failed-completer
MOVE-MANIPULATOR-with-tool-failed-completer
OPEN-MOVEABLE-FIXED-UNIT-failure-completer
CLOSE-MOVEABLE-FIXED-UNIT-failure-completer
DETACH-THREADED-FASTENER-failure-completer
INSTALL-REPLACEABLE-UNIT-failure-completer
REMOVE-REPLACEABLE-UNIT-failure-completer

Notice that there is more than one expander for the MOVE command. The reason for this is that there are different intents as to why the arm is moved somewhere, and by making these intents specific, there is not a need to go through some kind of a backtracking mechanism as the plans are developed. You know what you want to do, and that intelligence is represented in the various expanders.

Stage Two uses two tables to help it form the details of the commands it sends. The first is the action information table, the format of which is laid out below.

```
(deffacts action-relevant-information-table
;(<ACTION>, <RTC-SKILL>, <OBJECT-SUBTYPE>, <TOOL-CLASS>)

(attach-threaded-fastener, SCREW, bracket-screw, screwdriver-A)
(attach-threaded-fastener, BOLT-1, MEB-screw, wrench-B)
(attach-threaded-fastener, BOLT-1, door-latch-screw, wrench-A)
(attach-threaded-fastener, BOLT-2, bracket-screw, wrench-C))
```

The table is composed of entries, each having a parent class (action) and the object subtype to which it is applied. Also included are a corresponding command name for an RTC command which can be used for this purpose and the class of tool to be used in doing the operation. Notice different skill-tool combinations can be used on the same object subtype. This entry information is selected when the scheduling heuristics choose an action-unit, and the choice is added to the action-unit at that time.

The second table used by Stage Two holds semantic descriptions of points in space and is used to direct the arms to specific locations. A portion of the table is included below.

```
(deffacts spatial-lookup-table
   (approach-point BOLT-1 MEB-screw-1
      upper-left-MEB-screw-approach upper-left-MEB-screw-approach)
   (approach-point BOLT-1 MEB-screw-2
      upper-right-MEB-screw-approach upper-right-MEB-screw-approach)
   (approach-point BOLT-1 MEB-screw-3
      lower-left-MEB-screw-approach lower-left-MEB-screw-approach)
   (approach-point BOLT-1 MEB-screw-4
      lower-right-MEB-screw-approach lower-right-MEB-screw-approach)
   (approach-point BOLT-1 MEB-screw-11
      upper-left-MEB-screw-approach upper-left-MEB-screw-approach)
   ...)
```

For each skill and each object to which it can be applied, there is an entry in this table holding the associated semantic-location of where to move the arm in order to begin the operation, and where the arm will be left when it is completed. When commands are passed to the other components of TIPS (CENTeR and Audrey), the semantic information is translated into physical positions and joint-angle information. The task planner is not concerned with numeric values, only the semantic meaning associated with those values.

4.2.5 Example Execution

The example that follows is based on elements of the recent Solar Max satellite repair operation, undertaken by the crew of Shuttle Flight STS-11. In this scenario, an electrical component called the MEB has been diagnosed as being faulty. The MEB is fastened by four screws and further supported with a mounting bracket, which is fastened with two screws. Two cable connectors plug into the MEB, and the entire assembly is behind a door.

A high-level directive is given to RMS to replace the faulty component. Stage One expanded this directive into a plan of approximately 26 steps, involving disassembly of the satellite, the repair operation, and re-assembly. The repair plan is represented as a semantic network of action-units and is input to Stage Two from Stage One for execution. The following commands are an excerpt from the overall satellite repair sequence.

The action-unit below has been selected for expansion by the USE-TOOL-BEING-HELD heuristic, and has had appropriate robot information instantiated.

```
(defschema action-unit-87
   (exec-status readyed)
   (action attach-threaded-fastener)
   (object BRACKET-SCREW-2)
   (skill BOLT-2)
   (tool WRENCH-C-1)
   (arm RIGHT-ARM)
   (approach RIGHT-BRACKET-SCREW-APPROACH)
   (expansion-level 5)
   (comes-before action-unit-91)
   (comes-before action-unit-92)
   (comes-after action-unit-85))
```

The intent of this action-unit is to tighten a screw with the arm and tool specified. The robot has two arms, and one wrench of the type needed for this operation. The right arm holds the specified tool and the left arm holds a wrench of a different size. The following dialog is from a run of the planner with the kinematic simulator.

```
182 | From RMS
Move RIGHT-ARM along path:
(MEB-BASEBOARD-APPROACH RIGHT-BRACKET-SCREW-APPROACH)
```

Here a MOVE-MANIPULATOR expander commands the right arm to move to the approach point for the screw. The semantic names of the path elements are represented as a position and an orientation in task-space.

```
183 | From SIMULATOR
Failure Completer Issued to RMS
```

On simulating the command, the arm is discovered to be unable to reach the desired end-effector position. This information is received by a failure-completer, and the following actions take place. The action-unit which was made active, action-unit-87, is now stripped of the robot-specific information with which it was instantiated, making it the equivalent of all remaining action-units which have not been achieved. A slot which prohibits the use of the right arm is added to the action-unit, to prevent its being tried in this manner again. Since no action-unit is being executed, another is chosen from those available. Action-unit-87 is again selected, this time by the PICK-UNCLUTTERED-ARM-WHEN-TOOL-REQUIRED heuristic, and instantiated as follows. Though the same action-unit was chosen twice in a row, the planner treats it as a new object, and issues the following commands.

```
184 | From RMS
Move RIGHT-ARM along path:
(MEB-BASEBOARD-APPROACH HOME-RIGHT-ARM TOOLBOX-INTERPOSITION
WRENCH-C-1-TOOLBOX-APPROACH)
```

A MOVE-MANIPULATOR expander commands the right arm to move to the toolbox. A success message was issued, and a success-completer made the proper updates to the knowledge base, specifically that the arm is in a different location (success completers are not printed in this listing). As the overall task of the action-unit has not been achieved, the expanders examine the state of the world again to prescribe the next command. This process is repeated for each command in the remainder of this sequence.

```
185 | From RMS
Put WRENCH-C-1 with RIGHT-ARM

186 | From RMS
Move RIGHT-ARM along path:
(TOOLBOX-INTERPOSITION HOME-RIGHT-ARM)
```

The right arm, which holds the only available tool for this action, stores the tool and moves to a position which does not obstruct the workspace.

```
187 | From RMS
Move LEFT-ARM along path:
(TOOLBOX-INTERPOSITION WRENCH-B-2-TOOLBOX-APPROACH)

188 | From RMS
Put WRENCH-B-2 with LEFT-ARM

189 | From RMS
Move LEFT-ARM along path:
(TOOLBOX-INTERPOSITION WRENCH-C-1-TOOLBOX-APPROACH)

190 | From RMS
Get WRENCH-C-1 with LEFT-ARM
```

The left arm, which was holding another tool, is commanded to move to the toolbox, store the tool, move to another area in the toolbox region, and acquire the correct tool. Remember that each command is issued by an expander and that a completer accepts the status of the commands and makes the appropriate updates to the knowledge base.

```
191 | From RMS
Move LEFT-ARM along path:
(TOOLBOX-INTERPOSITION HOME-LEFT-ARM MEB-BASEBOARD-APPROACH
RIGHT-BRACKET-SCREW-APPROACH)

192 | From RMS
Bolt BRACKET-SCREW-2 with LEFT-ARM
```

Finally, as the left arm has been readied, it is commanded to move to the approach point for the bracket screw and tighten the bolt. Notice that the number of commands issued is based on the state of the world and what has to be done to ready it for the prescribed action to take place. Messages 185 through 192 all deal with the later instantiation of action-unit-87. The action-units provide a scope over which the planner considers what commands are executable at a given time.

This example run results in a repair plan of 120 steps, which RMS was able to generate in under one minute.

4.2.6 Specifications

4.2.6.1 Implementation

RMS is a system of about 120 rules written in the ART expert system environment. The current version of RMS, configured for the satellite repair scenario described in this publication, is about 15% control and inter-program communications rules, 35% expanders and completers, 10% heuristics, 25% NOAH data-expansion rules, and 15% Stage One support rules (constraint rules and NOAH critics). RMS is fully integrated with Audrey and CENTeR, through communication rules mentioned above and about 100 lines of supporting LISP code.

4.2.6.2 Characteristics

There are three primary advantages of the RMS task planner architecture described in this publication, each with regard to the domain of space-based robotic servicing.

4.2.6.2.1 Adaptability.

Unlike factory settings, exact knowledge of the target environment is not available until the service procedure begins. The expander-completer mechanism was developed in the course of this work and has been demonstrated as a method to cope with localized failures in a task-space without the need for a full-scale re-plan. The commands are generated in such a fashion that recovery is done as a part of the normal planning process.

4.2.6.2.2 Speed.

The architecture supports rapid plan generation. The hierarchical separation defined in the two stages of planning leaves decisions on specific details until the end of the process, thereby cutting down on the size of the planning space. The method for generation of Stage One plans was chosen because of the speed it exhibits in such domains. Further, the expander-completer mechanism in Stage Two works very rapidly, reactively generating commands faster than the robot can execute them. RMS does the overall satellite scenario described above in under a minute, producing a final sequence of some 120 commands.

4.2.6.2.3 Operator Guidance.

Finally, the architecture provides hooks in the planning process by which a user can guide the planning. Since Stage One deals with what overall operations are to be performed in the task-space, the operator can examine and edit the output should it be necessary. The hierarchical separation provides for a convenient place for an operator to add knowledge to a plan which is incorrect because of an inaccurate knowledge base. The operator also can guide the dynamic planning of Stage Two in situations where an operator's opinion is desirable, using the interface features of the Audrey simulator. An example is that of needing to do a pose change during a move operation. Prior to CENTeR integration, the system was configured to query the user for pose flip information to prevent risking a pose change where it might interfere with the task-space. Run-time modifications due to user suggestions are a desirable feature for such a space-based system where the human is acting primarily in a supervisory role.

RMS can be used as a command dispatcher for a robot repair operation. It also can be used as a tool for simulating a repair operation *a priori*, or as an advisor to a teleoperator or an astronaut. The architecture supports input and output at different levels, such as high-level directives, mid-level task steps, and low-level subsystem primitives, so that only applicable portions of a particular problem need be addressed. The architecture makes a conceptual separation of figuring out what to do from the specifics of how to do it, an approach not unlike that of a human.

4.3 THE GROSS-MOTION SPATIAL PLANNER

4.3.1 Introduction

The CENTeR spatial planning system provides automatic gross motion paths for manipulator arms in the JPL Telerobot Testbed. This spatial planner can be used by the teleoperator for moving the arm to specified configurations or for making safe pose changes automatically. Also, CENTeR is fully integrated with the TIPS task planning system to provide automated task and spatial planning capabilities.

4.3.1.1 Definitions

Before describing CENTeR, we will present some terminology used in spatial planning.

Spatial planning for a robot arm is the design of paths between given starting and ending robot configurations which avoid collisions with the objects in the workspace. *Gross motions* are those which affect the entire arm, whereas *fine motions* concentrate mainly on manipulating the end-effector. The CENTeR spatial planner finds collision-free gross motion paths for the manipulators. Using a description of the workspace, it generates and uses a mathematical graph of the object-free areas.

CENTeR is a geometric spatial planner rather than a trajectory planner or tracker. *Trajectory planning* means determining the desired position, velocity, and acceleration of the arm. *Trajectory tracking* means considering the actual versus the desired path. *Geometric path planning*, or *geometric spatial planning*, means finding the intermediate collision-free positions for the arm.

There are two ways to view the space in which the robot moves. One is *task-space;* the three-dimensional space about us. The other view of the space is as *configuration space* in which each point gives an angular position for each joint. Since the PUMA is a six-DOF manipulator, its configuration space has six dimensions.

The CENTeR spatial planner reasons in configuration space. The advantages of using configuration space over task-space for spatial planning are:

- In configuration space, a path is found which is collision-free for the entire arm. Paths generated in task-space are collision-free only for the tip of the end-effector.
- Working in configuration space allows the pose information to be expressed naturally. In task-space, pose information must be considered explicitly and safe pose changes are a constant concern.
- In configuration space, the orientation of the end-effector at each point along a path is a natural part of the configuration information. However, in task-space this orientation must be considered separately.

A configuration of the arm is said to *collide* with an obstacle if that configuration is prohibited by the obstacle. The set of configurations which cause no collisions is called *free-space.*

In CENTeR, we build a database which is a representation of the collision-free part of the configuration space of the arm. To explain why we took this approach, we shall review the current state of the art in spatial planning.

4.3.1.2 State of the Art

Spatial planning is surprisingly more difficult for a robot than for a human due to the size of the search space.

4.3.1.2.1 Current research.

In order to concentrate on a problem of reasonable size, most research in spatial planning centers on robots which can be modeled as points in a two- or three-dimensional space. That is, most research papers concern either a mobile robot, typically modeled as a point in two-dimensional space, or a two- or three-DOF arm, typically modeled as a point in two- or three-dimensional

configuration space. The work being done with such arms usually involves the use of sensors and allows moving obstacles in the workspace.

So far, the systems developed for these two- or three-DOF arms are not extensible to a six-DOF arm. For example, consider the work currently being done by Edward Cheung and Vladimir Lumelsky at Yale [G1]. In this work, an arm with infrared sensors can move between two points in a unknown workspace by following a straight-line path with modifications when obstacles are sensed. The modifications are developed by constructing a configuration space graph of the obstacle and following tangent lines around the obstacle until the original straight-line path is met again.

There are two problems with using the Yale approach in our work. One is that it often places the upper arm very close to an obstacle; if there is a minor sensing error, collisions are likely. The other problem is more substantial. The Yale robot had two-DOF, so its configuration space is two-dimensional. Thus objects may have tangent lines. However, if we extend to only three-DOF then there are tangent planes, and the direction to proceed must be selected intelligently. This poses a major problem in only three dimensions. If we should attempt this approach with even just four-DOF, the computations would become prohibitive.

At the IEEE Robotics Conference in April of 1988, there was no discussion about how to do spatial planning for a six- or seven-DOF arm. Two speakers erroneously stated that spatial planning for an arm in a fixed workspace had been solved; their work concerned other workspace types and two-DOF arms. Although there were solutions for spatial planning for a six-DOF arm in a fixed workspace, they were efficient only in spaces containing few obstacles. For cluttered spaces and a six-DOF arm there was no existing practical solution; that is, no solution that would find paths in real time taking all six dimensions into consideration. It is that practical solution for fixed cluttered spaces which we have found.

4.3.1.2.2 The Lozano-Perez system.

The main system we studied was developed recently by Tomas Lozano-Perez at MIT. CENTeR is similar to the LP system in that it requires a fixed workspace and that a database describing the collision-free portions of the workspace is preprocessed.

The reference [G3] gives the details of his approach. For two- or three-DOF arms, the Lozano-Perez method of representing regions is efficient for searching for a path. However, as the dimensions increase, the time required to find a path increases exponentially. The phrase *curse of dimensionality* refers to the fact that the worst-case time bound in any general motion planner is exponential in the number of DOF. Thus the Lozano-Perez system had to be modified to be applied practically to a six-DOF arm. The modification is to use a bounding volume for the end of the manipulator. This volume is a simple conservative containment of all of the possible positions of the last three links. Most of the path is planned for the first three links and the bounding volume. This strategy may fail to find a path, but Lozano-Perez found it to work in many of his test work-spaces.

The main reason we did not use the Lozano-Perez system is that the use of a bounding region for the last three links would not work in much of the telerobot testbed; the end-effector will be close enough to obstacles to have the bounding volume collide with them in many arm positions. If all six joints are processed using intervals of 20 degrees, there are almost five million possible

leaves on the interval tree. This is both computationally inefficient and unnecessarily detailed for areas away from objects.

We have developed a system which considers each of the six links and is fast enough to be useful.

4.3.2 CENTeR Overview

The CENTeR system (figure 7) consists of two parts: the off-line portion which builds the database model of the free-space for each arm, and the on-line path finder. This system will work in any relatively static workspace containing one or two manipulator arms of any number of DOF.

The graph generator uses a database of the workspace to generate the graph, and the path finder uses the graph to generate paths as they are needed at run-time. The graph generation code depends upon the number of DOF of the arm and the number of angles per joint being used in the graph. However, the rest of the code is independent of these variables, and all of the code is independent of the particular workspace under consideration.

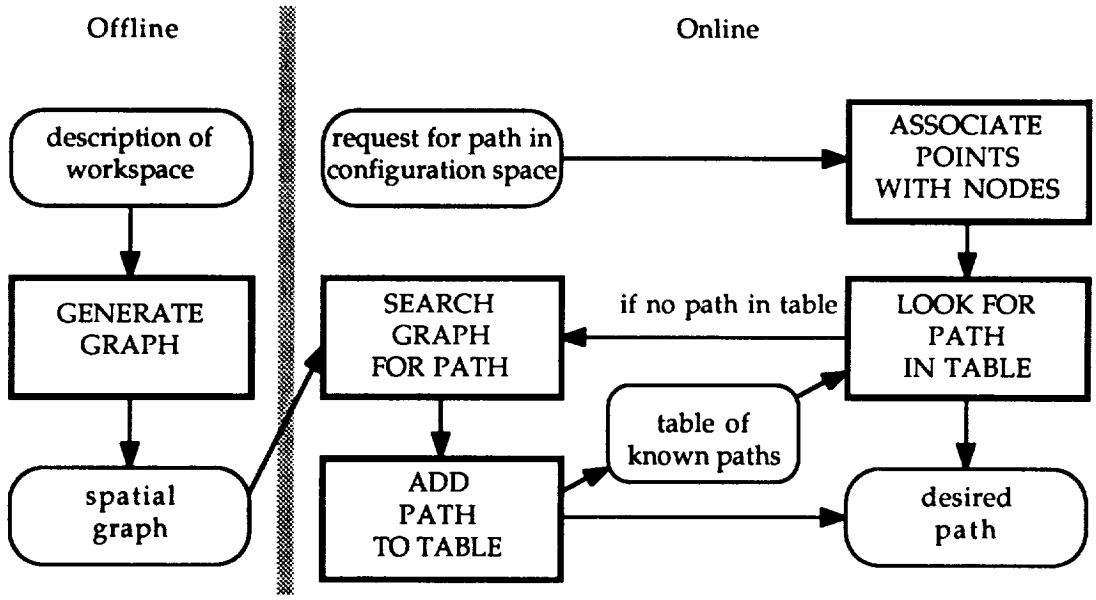


Figure 7. CENTeR Overview.

The average time required to generate a path on-line is less than four seconds. Thus for the telerobot testbed, the CENTeR spatial planner can generate paths much more quickly than they can be executed.

We have integrated CENTeR with the rest of the TIPS system for the satellite repair scenario. The user interface allows the operator several ways of using the spatial planner:

- to perform motions to operator-specified configurations
- to perform automatic safe pose changes
- to perform the gross motion tasks determined by the task planner — either those particular ones selected by the operator, or all of them

4.3.3 The Graph Generator

A graph of the free-space is generated for each arm. Each graph is built using a fixed number of selected angles for each of the joints of the robot. The configurations which are combinations of these angles are checked for collisions, and the collision-free ones form the nodes for the graph. Edges represent collision-free linearly interpolated arm movements between nearby configurations.

In the satellite repair scenario, five angles were selected for each of the six joints of the PUMA arm. Thus there were five to the power six, or 15,625, possible graph nodes, each with from six to twelve possible neighbors.

4.3.3.1 Description

The input to the graph generator consists of the number of joints in an arm, the selected positions for each joint, and a calibrated constructive solid geometry (CSG) model of the workspace. In a CSG model, objects are represented by unions, intersections and differences of geometric shapes. Their locations are given in terms of three rectangular dimensions and three orientation factors.

We begin by looking at some number b of positions for each joint (the end-points of the joint range and b - 2 evenly spaced positions within the range). For a PUMA arm, this gives us b to the power six initial points to consider, or 15,625 if b is five. Each point is tested for collisions.

Each of these points has between six and twelve possible neighbors, points which differ by one step in one coordinate. For each pair of collision-free neighbors, the straight-line path between them is checked for collisions. Notice that along such an edge, only one coordinate varies; that is, only one joint in the robot arm changes.

The output of the graph generator is the configuration space graph of free-space formed by the collision-free points and edges. The graphs are generated off-line due to the time required for collision detections.

4.3.3.2 Data Structure

The graph is represented as an array of adjacency lists. Each index number for the array corresponds directly to the arm configuration for the related node. To explain this correspondence, let us assume that five values per joint have been used, since such a graph is found to be sufficient for the satellite repair scenario. Since the PUMA has six joints, there are 15,625 (five to the power six) possible graph nodes. The array is indexed from 0 to 15624; the *i*th array entry is an adjacency list or is nil if the corresponding node is not collision-free.

To figure out the index which corresponds to a given arm configuration, we start by specifying the configuration by a list of the angle values of each joint, such as (90, -90, 180, -45, 97, 90). For each joint, there are 5 predetermined angles being considered; we can call them angles 0, 1, 2, 3, and 4. For the first joint, 90 degrees is the value of angle number 1; for the second, -90 is the value of angle number 2; for the third, 180 is the value of angle number 1. We can write the list as a list of angle numbers: (1 2 1 3 0 1). Thus each possible graph node corresponds to a unique six-digit base 5 number which in turn corresponds to a unique base 10 number between 0 and 15,624. For example, 121301 base 5 equals 4576 base 10. (1*1 + 0*5 + 3*25 + 1*125 + 2*625 + 1*3125 = 4576.) So the adjacency list for the configuration (90, -90, 180, -45, 97, 90) is stored in array location 4576.

4.3.3.3 Implementation

An Ada program using RTC collision detection software is used to generate a spatial graph. The main idea of the program is: Given a definition of the graph and a database representing the workspace, check each node (arm position) and edge (path between two adjacent nodes) for collisions and create a file containing the resulting information. The resulting output file is a list of each node, its collision status (true or false) and its corresponding adjacency list. Details as to how this information is stored can be found in the description of text_file_mgr.

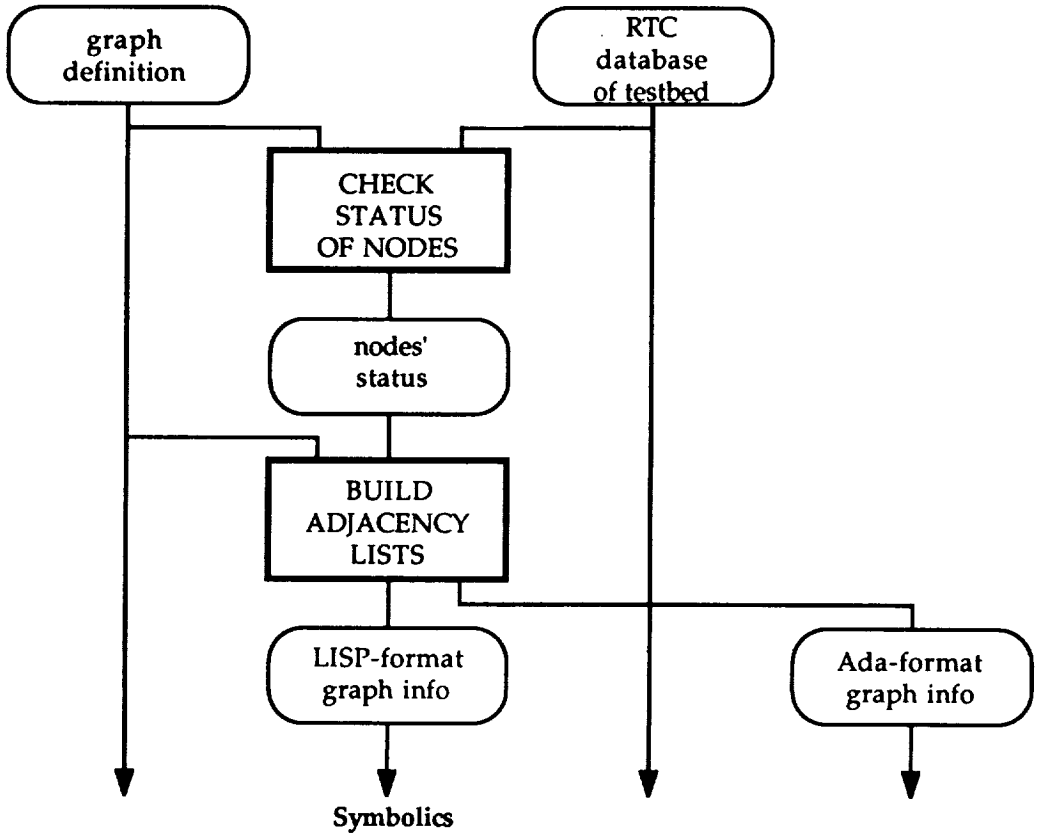


Figure 8. Graph generator node checking.

The program is separated into two main tasks: a) check all nodes for collisions, and b) check remaining edges for collisions. The accompanying diagrams show how these tasks are achieved (figures 8 and 9). Due to problems with using RTC's system and software, the main tasks are separated into sub-tasks. A great amount of memory space is required to run RTC's collision-detection software and to hold the main data structure. The main data structure is not needed when nodes or edges are being checked for collisions, but it is needed to find/update the adjacency lists. Therefore the tasks of checking for collisions and maintaining adjacency lists are performed separately to prevent a possible system crash due to memory overload.

The implementation of checking nodes is separated into two sub-tasks. RTC's database representing the workspace and a definition of the spatial graph (i.e., number of angles, number of joints, etc.) are given for the first sub-task. The resulting output is a text file containing a list

of nodes (the one-dimensional index number) followed by either a TRUE or FALSE, where TRUE indicates that the corresponding arm pose is free of collisions. RTC's collision detection software is used to check every arm position represented by the graph to see if there are any obstructions. The second sub-task uses the graph definition and information from the file produced by the first sub-task to find the corresponding adjacency lists for each collision-free node. An Ada file and LISP file containing complete graph information are produced. A detailed description of these files is described in the text_file_mgr description.

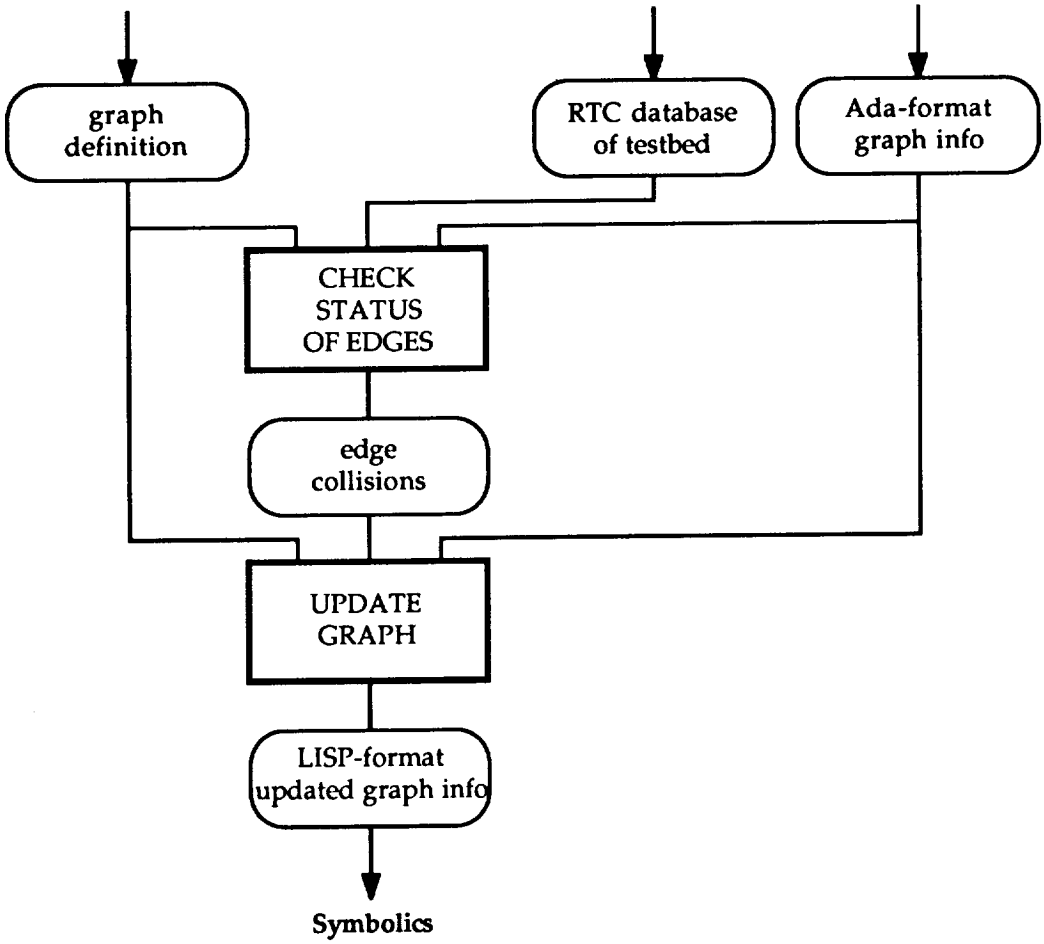


Figure 9. Graph generator edge checking.

The implementation of checking edges for collisions is also implemented as two steps. The first step has three main inputs: a) the final Ada file of graph information created when nodes were checked, b) the RTC database representation of the appropriate workspace, and c) the definition of the spatial graph. RTC's collision detection software is used to check all paths between adjacent nodes. The resulting output is a text file containing a list of node pairs (the one-dimensional index numbers) which indicate that a collision occurred on the path between the two nodes in each pair. This resulting file is used as an input for the second step along with the graph definition and the resulting Ada file created by the check-nodes task. The adjacency lists are updated in the Ada file by eliminating the appropriate nodes from the appropriate adjacency lists. The final outputs are a LISP file and an Ada file both representing the updated

information. These files are of the same form as those produced by the check-nodes task. Once a LISP file has been created, the file can be transported to a LISP environment where the other tasks of CENTeR operate.

4.3.3.4 Graph Sparsification

The generated spatial graph is searched to find paths through the workspace. If we have a space in which the graph is too sparse to allow us to find a path, we may increase the number of values per joint to generate a denser graph. However, this may create a large number of unnecessary graph nodes representing sparse areas of the workspace. When this occurs we can use a graph-modification procedure to strategically eliminate some of these unnecessary graph nodes. To describe a method for eliminating nodes, we first describe a method used for a two-dimensional graph, accompanied by diagrams (figures 10 and 11). Then we explain how the method can be applied to an n-dimensional graph.

The two-step algorithm for a two-dimensional graph:

```
for every node in the graph
   if the node is connected to the maximum possible adjacent nodes
         mark the node
for every node in the graph
   if the node is marked and the node has four adjacent marked nodes
         remove the four adjacent marked nodes
         create connections from the node to the removed nodes'
               adjacent nodes
         unmark all new adjacent nodes
```

The same algorithm can be applied to an n-dimensional graph. Instead of marking nodes having the maximum possible number of adjacent nodes, mark those which have m adjacent nodes, where m is ≤ maximum possible adjacent nodes. Also, for modifications, search for a marked node having m marked adjacent nodes (remove the marked adjacent nodes).

For this implementation in CENTeR, the main data structure representing the spatial graph is used as input and a modified graph is generated as output along with a text file containing the updated graph information. The main data structure representing the spatial graph is also used to represent a modified graph, where the possible node status values are:

nil	obstructed arm position
t	collision-free arm position
blue	collision-free arm position and has m collision-free adjacent nodes
green	collision-free arm position, which was previously blue and had m adjacent blue nodes

Modifications are made by updating the appropriate adjacency lists. After all modifications are made, the graph information is stored in a LISP file in a similar form to that of an unmodified spatial graph.

Modifications can be performed on a spatial graph to make the search space more sparse, thus decreasing the time it takes to search for a path to a goal position. With this algorithm, nodes will be mainly removed from areas where there are few or no obstacles, so that unnecessary

nodes are removed. Modifications can also be made between the two main runs of the graph generation program (first run checks nodes; second run checks edges). Since the number of edges will be decreased, the time it takes RTC's software to check edges for collisions will also be decreased.

4.3.4 The Path Finder

The other part of the spatial planner is the on-line path finder. The input to the path finder consists of three arguments: an indication of which arm is to be moved, the current configuration of that arm, and a set of acceptable goal configurations. The reason that the input contains a list of goals instead of a single goal is that for a desired position and orientation of the end-effector, there are up to eight configurations of the arm which will achieve the objective. The subset of these configurations which are kinematically possible for the arm forms the goal list.

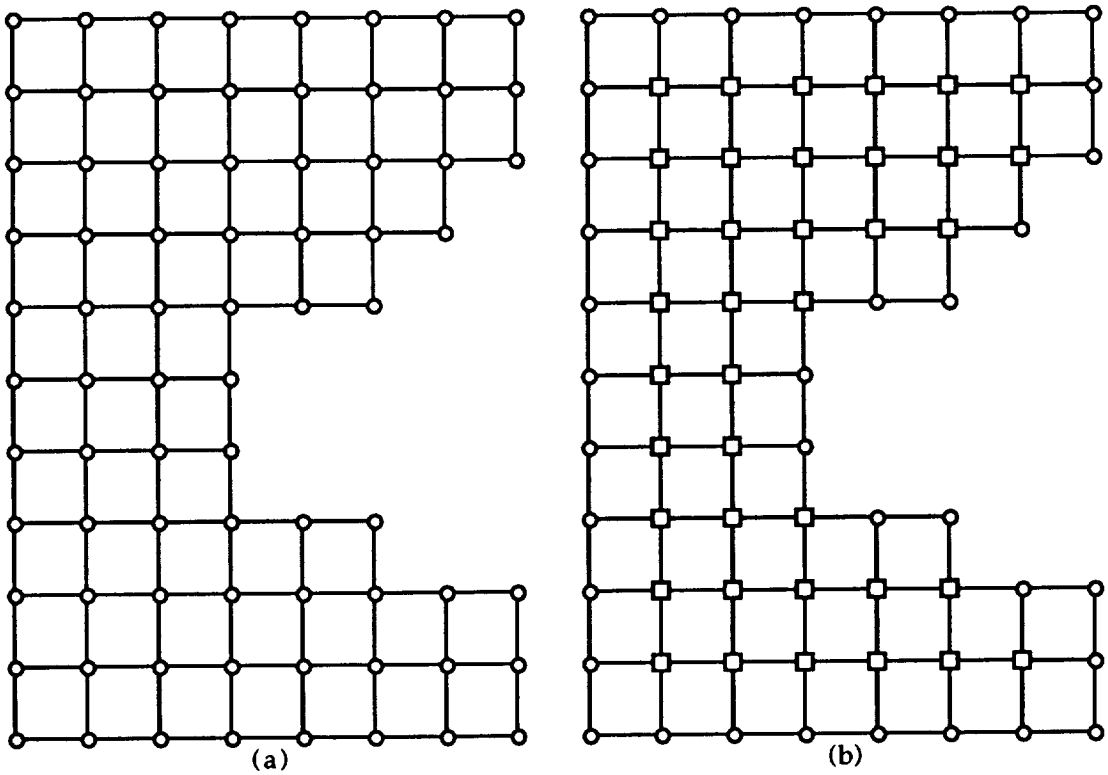


Figure 10. (a) Initial graph and (b) graph sparsification step one.

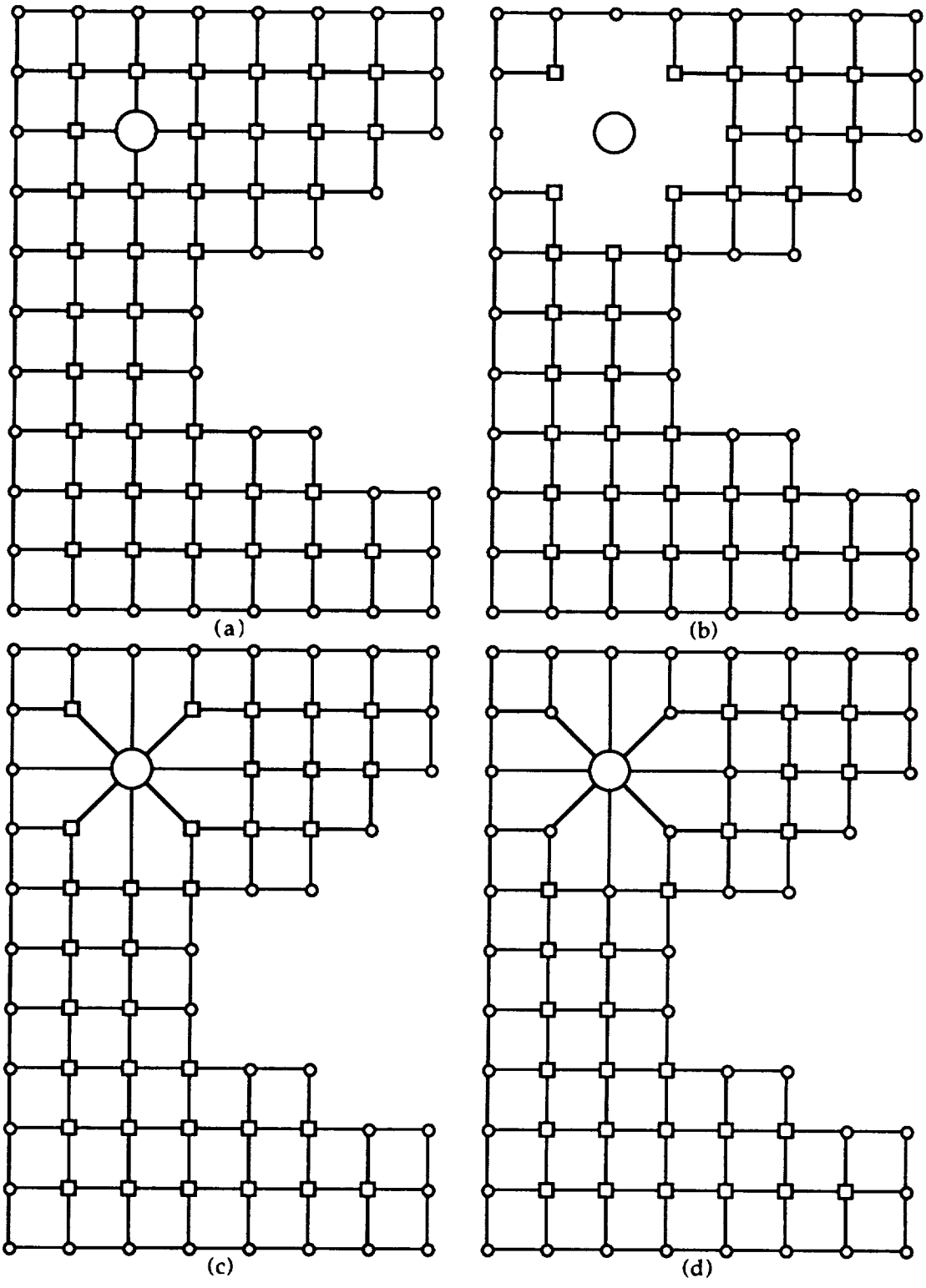


Figure 11. Graph sparsification step two.

When the TIPS system is integrated into the testbed, the list of arguments for the path finder will include an indication of the bounding region for the object in the end-effector and the current conditions of the variable workspace objects. These additional arguments will determine which graphs or graph parts to use.

The path request may come directly from the operator or from the automated task planning system. The output path is expressed as a sequence of configurations to be connected by linearly interpolated joint motion. This path is then passed to the other telerobot subsystems to be implemented. If there is no path to the desired goal, then a descriptive message is sent to the task planner or the operator.

There are three parts to each path: move to the graph; move along the graph; move off of the graph to the goal. There is a procedure for moving between a path end-point and a graph node. It finds the graph node closest to the point such that the interpolated motion between the two is collision free. Since we did not have on-line collision detection in the development of this system, this part has not been thoroughly developed.

To search the graph to find an acceptable path, we use an A* search algorithm [G4] which has been modified to accept a set of possible goals. Two heuristics are used. One associates with each search node the actual length of the path from the start node to that node. The other estimates the remaining distance to the closest goal.

A path storage and look-up facility is also provided for run-time speed enhancement. Since the path finder is so fast, path look-up is not being used in the integrated system. It is described here in case a situation should arise in which it would be useful.

If a requested path or its reverse has already been generated it is not regenerated. This provides time savings when arm motions are repeated often, such as the motion from the toolbox to a work area on the satellite. Given a request for a path, first the tables are queried to see whether or not the path has already been generated. If the path is not in the tables, it is generated and inserted into the appropriate table. There are separate tables for paths for different arms and for different configurations of movable workspace objects.

If the workspace is extremely cluttered, it may be necessary to preprocess the path look-up tables. All probable needed paths may be generated and a path look-up procedure may be used during operation if time constraints require more speed than the path finder can provide. However, based upon our current research, it is hard to imagine a situation in which this would be necessary.

4.3.5 Work Space Assumptions

The task-space for the initial JPL Telerobot Testbed consists primarily of a Solar Max satellite, two manipulator arms, and a tool box. This is the scenario used for the development of the integrated TIPS system, but the spatial planner in its present form can be used with any fixed workspace with one or two manipulator arms. The arms move independently, and when one is moving the other is placed in a known home location. The spatial planning code could be easily adapted to a workspace containing additional arms by including related free-space graphs.

The robot frame is considered to be fixed relative to the workspace, and this relative position is known in advance. A small fixed number of known variable objects, such as a door, may be

present in the workspace if each has a small fixed number of known positions in which it normally occurs, such as opened or closed. Each such object must remain in a known position during any planned gross motions. For example, the camera-arm motion will be restricted to a fixed region. Before invoking the spatial planner, the operator must return the camera-arm to this region.

A relatively static database was assumed in the design of CENTeR for several reasons. All current spatial planning systems for six-DOF arms, such as the Lozano-Perez system, make this assumption; we decided to develop a system which adapts the state of the art to a real-life space scenario before working on a more advanced system. Also, the graphics system and the collision detection software that we are using require that the workspace be static. We are hoping to purchase more up-to-date commercial graphics and collision-detection software soon.

But most importantly, there are space applications of robotics in work-spaces which are known a priori for spatial planning purposes, such as servicing the space station or servicing a satellite like the Solar Max or the Earth Observing System (Eos).

4.3.6 Usage

4.3.6.1 Direct Use by the Teleoperator

The teleoperator can use the spatial planner to perform motions between operator-specified configurations of the arm. One instance in which the operator may wish to invoke the spatial planner for a path is in changing poses for the arm. In CENTeR, pose flips do not come into play since the paths are generated in configuration space rather than task-space. However, when working in task-space, as the teleoperator does, one sometimes needs to change the poses for the arm. These pose flips can be dangerous since the arm may be extended toward an obstacle.

For example, the arm may be in an elbow-up pose with the end-effector near the satellite. To perform an operation, the arm may need to be in an elbow-down pose instead. The spatial planner can generate a path from one pose to the other which will keep the arm from colliding with workspace objects. In this manner, CENTeR can be used to execute a pose flip safely for the teleoperator.

4.3.6.2 Use with the Automatic Task Planning System

In automatic mode, given a request a path is found automatically and sent through the system to be executed. In monitor mode the teleoperator decides whether to use automation or teleoperation for each path. If teleoperation is chosen, then the teleoperator must inform the system when the motion is finished. In the fully integrated testbed, the results of a teleoperation will be gleaned from the hardware.

4.3.7 Specifications

4.3.7.1 Software Modules

4.3.7.1.1 Ada.

Aside from the driver programs, the Ada design is broken up into five parts, the following packages:

Graph_data

This package contains information necessary for the main data structure. This includes constants for number of joints, number of angles per joint and number of total nodes. The n-dimensional graph, representing configuration space, is mapped onto a one-dimensional array, where n is the number of DOF. Here, the main data structure called coll_graph is declared as a one-dimensional array of records, where the first slot of the record is c_free of type boolean to indicate whether the node is collision-free or not, and the second slot is an array which holds the node's adjacency list. The array is indexed from 1 to num_nodes. The index number of the array corresponds to the one-dimensional node number of the graph. A 0 (zero) in the adjacency list indicates a null or non-existent neighbor.

Index_translation

This package contains two procedures: joint_space to get the n-dimensional index number for a given one-dimensional index number, and graph_index to get the one-dimensional index number for a given n-dimensional number.

Adj_list_mgr

This package contains procedures for creating and updating adjacency lists. The adjacency list is an array of length max_num_neighbors(declared in graph_data). A 0 (zero) in the array represents a null or non-existent node. C_free_neighbors returns an adjacency list for a given node, where each node in the adjacency list is collision-free. Eliminate_nbr eliminates a given adjacent node (nbr) from a given node's adjacency list. This is achieved by replacing nbr with a 0 (zero) in the appropriate adjacency list.

Joint_angle_data

The actual angle values for each joint that is represented in the configuration space graph are contained in this package. A function for accessing this table is also in this package: ja_values returns an array consisting of six angle values for a given one-dimensional graph node. The joint-angle table has a set of default angle values for 5 angles per joint. These values can be changed when using RTC's Thread.

Text_file_mgr

Procedures responsible for saving information from the main data structure to text files are found in this package. Save_lisp_graph saves the information in LISP form for use in the Symbolics environment and Save_ada_graph saves the information in Ada form. The LISP file is a list containing lists on each line. The line number corresponds to the one-dimensional index number of coll_graph. The list on each line contains two elements. The first is a t or nil to indicate whether the node is collision-free or not and the second element is an adjacency list containing the one-dimensional adjacent node numbers. For CENTeR's data structure, the array is indexed from 0 to (num_nodes - 1). Therefore, a node in this file corresponds to node + 1 in coll_graph. The Ada graph has num_nodes number of lines, where each line number corresponds to the index number of coll_graph. The first element on each line is either a TRUE or FALSE to indicate collision-free or not. This is followed by each number found in the corresponding adjacency list. Get_ada_graph extracts information from a file created by save_ada_graph and uses it to initialize the main data structure.

4.3.7.1.2 LISP

The spatial planner LISP code consists of several modules.

The module **initialize** contains the code for loading the spatial planner. It also contains two functions:

initialize base-number (= number of angles per joint); dimensions-number (= number of joints; no. of degrees of freedom)

Result: initializes the path look-up tables to nil and initializes the global variables base, dimensions, and array-size.

init-graphs scenario ('SATELLITE is implemented now, 'EOS should be in the near future)

Result: Initializes the graphs that will be used for the specified workspace.

The module **point-converter** contains the function get-path which is invoked to find paths. The start and goal configurations are converted to their nearest graph nodes before look-up-a-path or find-path are called to search for a path. Several auxiliary functions are also included:

array-index value (in degrees); joint-number (between 0 and dimensions - 1)

Result: the number (between 0 and base-1) of the angle for that joint with that value.

nearest-graph-node angle-list (a list of actual angle values for each joint); arm-graph

Result: a base 10 index number for the collision-free node closest to angle-list.

translate node (1-dimensional index number or list of joint values); arm ("left-arm" or "right-arm")

Result: If the node is indicated as a one-dimensional index number, it is returned if collision-free. If the node is indicated as a list of joint values, the one-dimensional index number of the nearest node is returned. Otherwise, nil is returned. This function calls nearest-node.

closest-goal index (one-dimensional); goal-node-list (list of lists of joint values)

Result: the entry in goal-node-list whose nearest graph node is the index.

node-equal node1; node2 (lists of angle values in degrees)

Result: True if-and-only-if the differences for each joint are less than two degrees.

remove-nils old-list

Result: the list of the non-nil elements of oldlist.

get-path arm ("right-arm" or "left-arm"); start-node; goal-node-list (each node can be given as a one-dimensional index number or list of angle values in degrees)

Result: nil if no path exists; otherwise a path expressed as a list of lists of angle values in degrees. This function calls look-up-a-path in the lookup-tables module.

The module **lookup-tables** contains the code for storing and retrieving paths as they are generated:

initialize-path-lookup-tables no arguments

Result: sets the lookup tables to nil.

show-path-lookup-tables no arguments

Result: displays the contents of the path look-up tables; used for debugging and for illustrating the path storage facility.

check-for-a-path start (); goal-list (); arm-number()

Result: path stored in look-up table; or nil. If the reverse of a requested path is stored, it is returned in the correct order for this check-for-a-path request.

look-up-a-path arm-to-use (); start-node-index (); goal-node-index-list ()

Result: calls check-for-a-path; if no path was found in the look-up tables, then the variable *spatial-graph* is set to the appropriate graph, find-path is called to search the graph for a path, and the new path is inserted into the look-up tables and then returned.

The module **path-finder** contains the graph-searching code for generating paths:

in-search-graph-find-node my-index (a base 10 index number)

Result: the node in *search-graph* with the given index. *search-graph* is stored as an array of structures.

list-digits index

Result: a list of the digits of the base 5 number which corresponds to the given index.

goodness index; goal-index-list

Result: the sum of the distance from the start node and an estimate of the distance to the goal. The estimate of the distance to the goal is the minimum of the estimates of the distances to the goals, each

given by the sum of the absolute values of the differences in the joint steps of the two arm positions. (First change each base 10 index into a base 5 number and determine the sum of the differences in each slot.)

find-path start-index goal-index-list

Result: a path of the form (start, index, index, ..., index, goal) where start is start-index and goal is an element of the goal-index-list. Uses a modified A* search algorithm adapted from Nilsson.

The module **graphics-patches** contains the code for the interface between the spatial planner and the graphics system:

audreyii-move-manipulator PUMA-NAME; C6-POINT-PATH

Result: the Audrey function adapted to handle moves.

center-get-path PUMA-name start-joint-array end-joint-array-list

Result: the Audrey function adapted to put the arguments into the correct form for invoking the get-path function.

The module **user-interface** contains the code for the user to invoke the spatial planner and to find out information about the current state of the workspace:

name-a-place name ('anything); degree-angle-list ('(# # # # # #))

Result: stores the name of a configuration for future reference. Makes sure the list has 6 elements first. This named configuration can be used in the center-move and place-arm functions.

name-points-near-satellite no arguments

Result: for the demo, this names two configurations, '14 and '16, which put the end-effector in the same position and orientation, one with the elbow up and one with the elbow down.

in-tablep name ('anything)

Result: If the name has been stored, return the corresponding configuration. Otherwise, return nil.

set-center-mode mode ('monitor or 'auto)

Result: sets the global variable *path-monitor-present* to true if 'monitor and false if 'auto. This controls whether or not the tele-operation option box is presented upon path requests from the task planner or other sources.

center-move arm ('right-arm or 'left-arm); goal ('toolbox, 'satellite, 'home, or another named place)

Result: invokes the spatial planner with a legal goal list by calling audreyii-move-manipulator.

place-arm arm ('right-arm or 'left-arm); place (a list of joint angles in degrees or a semantic name)

Result: displays the arm in that place. Updates Audrey database, but not RMS database.

current-graph-node arm ('right-arm or 'left-arm)

Result: returns a statement of the base 10 index number of the nearest graph node to the current arm position, and tells whether or not the arm is actually at that node: sets the variable arm-graph appropriately, finds the list of current joint angles, finds the nearest graph node to the current configuration.

current-arm-position arm ('right-arm or 'left-arm)

Result: returns the angle values for the current configuration of the arm.

set-goal-list c6-point-path (base 10 index or named place); PUMA-name

Result: makes an array to hold the goal list.

c-free-point-from-list arm ('right-arm or 'left-arm); list (of configurations - each configuration is a list of angle values for each joint)

Result: the arm position of the graph node nearest the first list element for which a nearby collision-free graph node can be found; otherwise nil.

auto-move-manipulator no arguments

Result: sends the notices when the spatial planner begins and ends its thinking; calls center-get-path for a path; causes the graphics for the path to be displayed, and sends a completer to the RMS.

move-done no arguments

Result: puts the arm in a collision-free final position and returns a success completer to the RMS. Used when the operator chooses tele-operation and that operation is successful.

move-failure no arguments

Result: send failure completer to RMS.

manual-move-completion no arguments

Result: displays a menu for operator to select teleoperation result, either "arm near goal" or "arm near start"; handles the operator's choice by calling move-done or move-failure, respectively.

manual-move-manipulator no arguments

Result: invokes manual-move-completion.

auto-or-manual-choice no arguments

Result: displays a menu for the operator to select teleoperation or automation for a gross motion; handles the operator's choice by calling auto-move-manipulator or manual-move-manipulator, respectively.

The module **translations** contains several functions used in translating between different methods of referring to graph nodes:

arm-position-of-index index (base 10)

Result: a list containing the actual angle values for a robot arm.

index-of-arm-position angle-list (list of angle values in degrees)

Result: the one-dimensional index of the node with those joint angles.

get-1dim-index index (expressed in base base - other than ten)

Result: base 10 index number which corresponds to that index.

get-multi-index index (base 10)

Result: a list containing the digits for a base base number which corresponds to the index. This is the list of the angle indices for the index position.

The modules **file-management** and **sparse-spatial-graph** contain the code for initializing the graph arrays from the results of the collision detection programs and for sparsifying the graphs, respectively.

4.3.7.2 Performance Data

In the integrated TIPS system, removing the MEB from the satellite and storing it in the toolbox required 36 gross motion paths. The times for finding paths averaged 3.8 seconds, and ranged from 0.4 to 11.5 seconds.

The graph was built with five positions per joint, making the number of possible nodes in the graph 15,625. The time to check the graph nodes for collisions was about 12 hours using RTC collision detection software, which runs at 2 to 15 seconds per node.

4.3.8 Status

4.3.8.1 Original Goals

In designing the CENTeR spatial planner, we had six goals in mind. The system we have developed achieves each of these goals.

4.3.8.1.1 Suitability for integration into testbed.

A foremost consideration was the development of a spatial planning system which is compatible with the needs of the JPL Telerobot Testbed. CENTeR is capable of finding desired paths in a telerobot scenario quickly.

4.3.8.1.2 Speed.

The on-line path generation is very fast. In the satellite repair scenario, the average time required to generate a path is less than four seconds. Thus the CENTeR spatial planner can generate paths much more quickly than they can be executed, which means that the system will be useful to the teleoperator.

4.3.8.1.3 Ability to accept minor perturbations in the workspace.

Currently, the collision detection we use indicates a collision when the arm is within a certain tolerance of an object instead of only when a collision is occurring. Thus minor perturbations in the workspace can be tolerated without invalidating the graph database of the CENTeR spatial planner. If path requests are made to the correct destination, then the CENTeR system will find a collision-free path even if the workspace has been slightly altered or the calibrations were slightly incorrect.

4.3.8.1.4 Ability to handle carried objects.

In order to ensure that objects held by the end-effector do not collide with obstacles in the work-space, we generate additional graphs or graph overlays for each arm, each graph assuming a different bounding region for objects in the end-effector. Then requests to the path finder would include an indication of which bounding region is currently appropriate so that the correct graph will be employed. We should have, say, 4 graphs for each arm: one for an empty end-effector, and three for various sized bounding regions for objects in the end-effector.

We do not have the additional graphs now due to time constraints on using the RTC computer and the speed of the collision detection algorithm. We are looking forward to having fast collision-detection software dedicated to gross-motion spatial planning research in the near future.

4.3.8.1.5 Ability to accept variable workspace objects.

The telerobot work-spaces may contain a small number of variable objects. Such an object can have a small number of positions which are fixed and known in advance. Then for each position of a variable object, a graph or graph overlay for each arm can be generated. When a path is requested, the current position for such an object is passed as an argument so that the appropriate graph or graph part will be used.

For example, a door is a variable object with two positions: opened or closed. In performing most gross motions, it can be safely assumed that the door is locked into either the opened position or the closed position. If necessary, the door can also be modeled by a conservative bounding region which contains all of the possible intermediate positions. This model can be used when the door position is unknown. Thus at most three models for the door are required: the opened door, the closed door, and the bounding region for unknown door positions.

This modeling for a door is sufficient since the teleoperator will be able to handle arm movements in anomalous workspace configurations.

4.3.8.1.6 Adaptability.

The graph generator uses a database of the workspace to generate the graph, and the path finder uses the graph to generate paths on-line. All of the spatial planning code is independent of the particular workspace under consideration. In addition, all of the path finding and storage code is independent of the number of DOF and the number of angles per joint being used in the graph, and so can be used with any type of arm and any degree of granularity of the graph.

4.3.8.2 Features which Exceeded Expectations

4.3.8.2.1 Speed.

Originally, we had expected that the on-line path finder would be too slow to be used in real time. We had planned that for the testbed all necessary paths would be generated and stored before the demonstration runs. Then during the demonstration, all paths could be looked up. We are quite pleased that this will be unnecessary. Since it can be used in real time, the system will be of use to the teleoperator.

4.3.8.2.2 Sufficiency of the generated graph.

We had expected to have to add a number of additional points to the generated graph to enable access to work areas. However, the generated graph has been proven to be sufficient for the satellite repair scenario in our simulations. If a future workspace contains work areas that are not accessible by the spatial planner, we will generate graphs with more angles per joint. These graphs can be sparsified to eliminate unnecessary nodes from the work areas. Thus we will be able to maintain workspace independence in our code because we have control over graph granularity and the ability to sparsify.

4.3.8.2.3 Direct use of the spatial planner by a teleoperator.

Initially we required that the operator could use the spatial planner only through the task planner. This restriction has been lifted. The spatial planner can be used by the operator either independently of the task planning system or in combination with task planning and as much teleoperation as is desired.

4.3.8.2.4 Spatial planner supports traded control.

Control between automatic spatial planning and teleoperation can be exchanged anywhere in the workspace. Originally it was expected that trading control would have to occur at places convenient to the spatial planner, but this restriction is not necessary since the graph nodes are dense enough that collision-free motion to a nearby node will be easy to perform.

Initially we also required that trading control from the operator to the spatial planner would not be allowed when the end-effector is holding an object which was grasped by the operator. However, if a bounding region for the object with respect to the end-effector is known to the spatial planner, then control trading may occur when operator-grasped objects are held.

4.3.8.3 Current Research Directions

4.3.8.3.1 Sensor feedback.

When we get dedicated collision detection software, then we will be able to experiment with enhancements which allow for sensor integration and moving obstacles in the workspace. We are also planning to research areas of sensor integration for non-graph-based systems. The use of sensors for updating the free-space graph, or in place of it, becomes necessary when the workspace is less predictable due to object motions. However, we do intend for the results of our research to be applied to six- or seven-DOF arms.

A graph-based system is sufficient in a fixed environment which is thoroughly known ahead of time, such as a satellite repair scenario, although sensors are useful for verifying the workspace model. If the environment is fixed but not known ahead of time, and if the work activities will be performed repeatedly, sensors may be used to develop a graph. Then the graph-based approach may be followed. Sensor feedback may be used in place of a graph in an unknown environment which is variable or which is visited only briefly. However, if the workspace is relatively fixed, it may be desirable to use a graph as a short-term memory of the robot's immediate surroundings.

One sensor-graph integration approach we are considering provides a sensor-based graph-update facility for handling a workspace in which things may move, and then freeze for our action. Changes would not require recomputing the entire graph, only the parts of the graph that are affected. For example if we are expecting a certain fixed workspace and the actual space is slightly different, the graph-update facility will allow us to use the automatic spatial planner without regenerating the entire graph.

If a look-up table is used, the listed paths could be time-stamped so that old paths which are no longer collision-free would not be used. A time stamp feature can be used on paths if workspace objects may vary in a non-predetermined manner. Then when a path is retrieved from the look-up tables, we first check to see when it was generated. If the graph has changed significantly since the generation of the path, the path's validity is checked. A path verifier could use time stamps to verify the parts of the path that need to be redone and ignore those which do not.

Our first step in the study of sensor feedback is to determine what sensor feedback would be useful to a spatial planner, and how it would be processed. Also, the mechanisms need to be developed for information to be passed from the sensing and perception subsystem to TIPS for use by a spatial planner.

4.4 THE USER-INTERFACE AND KINEMATIC SIMULATOR

4.4.1 Introduction

Audrey (figure 12) is a computer program which simulates the motion of devices by means of computer graphics. Objects are defined by size and shape; devices are defined as a linked series of objects for which particular motions are possible. Audrey accepts commands from both a user at the Audrey console and from external sources such as a remote terminal or another computer program. When commanded from an external source, Audrey returns a variety of useful information on the results of the commanded action. Audrey is able to utilize external sources of

knowledge in order to create a more realistic simulation. The external knowledge can be from either a computer program or from a human expert. Audrey coordinates requests for both information and action, based on the available resources in TIPS. Audrey routes requests either to the computer program or to the Audrey user if the computer program is currently unavailable. Audrey maintains the modularity of TIPS by modeling knowledge sources as either computer or human. This modularity allows for the replacement of any computer portion of TIPS with a human.

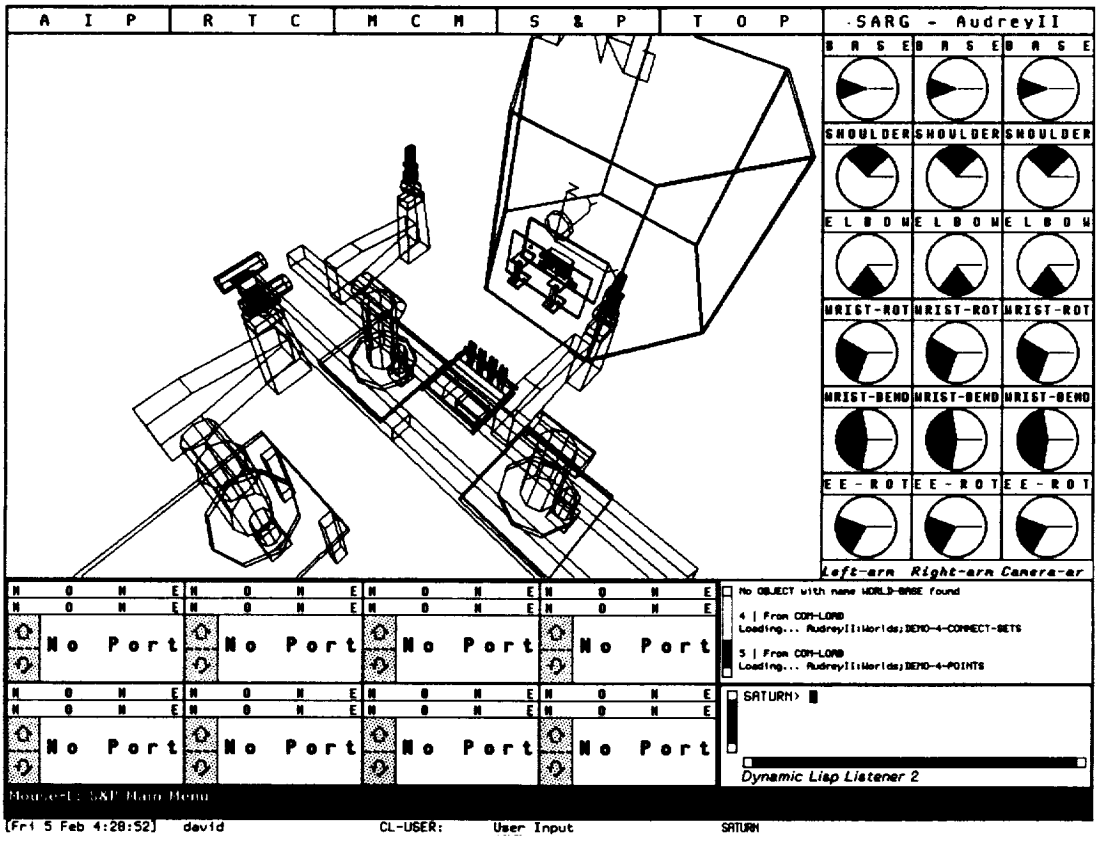


Figure 12. The Audrey screen.

4.4.2 Windows

Audrey is a window-based computer program which utilizes a mouse and menus for the majority of its input. The mouse is used to indicate screen objects to which actions are applied, and is used to move objects in the modeled world. Windows are used to separate conceptually different information, such as graphic line drawings from textual warnings.

4.4.2.1 Viewport

The viewport displays a perspective wire-frame rendering of the current world as seen through one of several cameras. By mousing the objects in the viewport, the user can command a variety of actions. The user can view the world from any angle and position by moving the camera that is currently generating the image which is seen in the viewport. The user can select a view from a variety of cameras. The viewport has the ability to perform graphic *hither-clipping*, a technique in which objects which are too close to the camera to be rendered correctly are deleted

from the scene. The use of hither-clipping allows the user to place the camera as close to an object as desired, even though that would require placing the camera within another object; the object which is in the way of the camera is then clipped from the scene and not drawn. The use of additional graphic techniques is limited by the speed of the Symbolics computer.

Processor limitations of the Symbolics LISP Machine preclude a complete graphic rendition of the world. For this reason, Audrey utilizes the simplest graphic representation possible. Audrey's graphic window, the viewport, outputs a wire-frame drawing with perspective, and can perform graphic hither-clipping, although with an order of magnitude decrease in performance from non-hither-clipped drawing speed. Audrey utilizes multiple bit planes for rendering its wire-frame scene. A single bit-plane is used for drawing those objects which cannot move, another is used for those which can move, but are not doing so at the time. The third bit-plane is for the object, or objects, which are being animated. This third bit-plane can be drawn on and erased from without disturbing the other objects in the world which have not changed position. A fourth bit-plane is used for combining the first three before the image is placed upon the screen for the user.

4.4.2.2 Tree Window

The user can browse through the database of objects by viewing the tree structure in the tree window. Viewing the tree structure aids the user in understanding the physical relationships between objects. Future implementations of the tree window will include the ability to view additional information about the objects and their connectivity, and the ability to prune and graft portions of the database.

4.4.2.3 Dial Box

The dial box window is a screen-based equivalent of a physical dial box. An actual dial box typically consists of six to ten dial potentiometers which can be twiddled by the user. The value of each of the potentiometers, which varies from zero to one, is read by the computer as the current value of some scalar and then scaled to within some target range. If the dial has an assigned relationship to the rotation of an object around its Z-axis, then value of the dial would be scaled to be between zero and 360 degrees. Audrey allows each dial to be connected to any of a variety of scalar values defined for the objects in the world.

Rather than use a graphic representation of a dial, Audrey implements a dial as a rectangle in which is displayed the current value along with adjustment arrows for moving the value higher and lower. Actual dials are not used because it is too difficult to track around a circle with a mouse without paying undue attention to the computer screen. Through the dials, which are called *arrow gauges*, the user can input exact values or adjust the current value up or down by clicking on the adjustment arrows. Each click modifies the current value by a definable resolution. Arrow gauges are clearly labelled with the name of the object and the scalar characteristic which is being manipulated.

4.4.2.4 Warnings

The warnings window maintains a complete record of warnings and messages sent to the user. This record can be reviewed at any time, except when a new warning is being presented. The warning record scrolls to the end of the list of warnings when a new warning is being presented. All warnings are time-stamped, and their origin is indicated by the sender.

4.4.2.5 Joint Gauges

Joint gauges are circular indicators which monitor all robot joints in the world. Joint gauges display the current angle of the joint as well as its singularity and physical joint stops. The joint gauge indicates violations of allowable values by displaying itself in reverse video.

4.4.2.6 LISP Listener

The LISP Listener allows the user a command-line access to the computer, and a complete interpreted LISP environment. The LISP Listener is a window which is provided by the Symbolics computer, and has run-time and interactive debugging.

4.4.2.7 Menus

The menu windows are Audrey's choice facility. Menu selections are remembered, and the item selected is presented as the default choice upon the next presentation of the menu. The user can modify the family, face, and size of the font used for the menu text to suit personal taste. All menu windows pop-up near the mouse location in response to either a user or a computer request.

4.4.3 Software Structure

Audrey's software is organized in several levels, which insures the necessary speed of operation and the robust error trapping required for a useable computer program. Internal routines are optimized for speed, with little error trapping; the graphics kernel and other internal commands are part of this set. Those commands which are to be called from external sources are designed to be more robust in error trapping while returning useful information under a variety of error and non-error conditions. Generally for each internal command there exists an external command. The correspondence between internal and external commands also exists for menu commands. Menu commands are to be used only by the user at the Audrey console, and are important for their effect, rather than for the information they return.

4.4.3.1 Graphics Kernel

A small kernel of graphics routines exists for drawing and erasing objects, and for moving objects from bit-plane to bit-plane. Many of these routines can be applied to either a single object, or to the object and all its descendents. An attempt has been made to make the kernel routines as fast as possible. Since the Symbolics is a general-purpose computer, as fast as possible is not as fast as desirable.

4.4.3.2 Internal Commands

Internal commands do not attempt to trap for errors in the arguments which are passed to them. Rather, internal commands are designed to be called from either a menu, which is guaranteed to pass the correct arguments, or from an external command which performs error trapping.

4.4.3.3 External Commands

The primary purpose of external commands is to be called from another program running on the same computer, perform error trapping, execute the requested action, and return any useful information about the result of the command. Information queries, such as those for the position of an object, are also handled through external commands. Should the mouse cease functioning

for any reason, the user at Audrey's console can operate the program completely from the keyboard.

4.4.3.4 Menu Commands

Menu commands form the majority of the user interface. Two methods of menu interaction are possible: Object-Action-Confirm and Action-Object-Confirm. In the first case, the user indicates an object for consideration by clicking on any vertex of the object as it appears in the viewport window. After indicating the object with which to work, the user chooses an action from a menu, and is asked to confirm his choice. This method of interaction is not always desirable, either because of personal preference, or because the object which we want to perform an action upon is not visible in the viewport window. The second method of interaction requires the user to select an action from a menu, then indicate the object to which the action is to apply. Execution of the action is made only after user confirmation.

4.4.4 Modeling

4.4.4.1 Objects

Audrey utilizes a simple scheme for modeling three-dimensional wire-frame objects. Objects are given case-insensitive names for external reference, as well as a *pretty-name* which is used for internal reference. Conversion rules maintain the traceability between internal and external object names. An example of a conversion rule would be the correspondence between the use of underscores (_) and the use of hyphens (-) in object names; underscores are prohibited in the names of objects used internally, and are converted to hyphens or spaces. An external reference to an object may be made with hyphens, underscores, or spaces, depending on the most convenient representation of the referencing system. External references are matched to internal names by allowing for differences in the use of hyphens, underscores, and spaces. Integer indices can also be used to refer to objects in place of object names.

Audrey also maintains information about the general category, the object's *type*, in which an object resides. This type is used when an object must locate another object of a certain type in order that relative data may be determined. For instance, when a robot end-effector receives a request for motion, the end-effector determines its current position by locating the position of a parent object which has the type 'PUMA-SHOULDER-FRAME. The parent object with this type is the object from which we determine the relative location of the end-effector.

Each object contains a reference to its parent object. The *parent* slot is a pointer to an object in the database on which the first object is physically dependent. When the parent object moves, it informs its *children* that they too must update their position. An object's children are simply pointers, contained in a list, to the objects which are physically dependent upon the object. An object can have many children, but only one parent. This is the standard tree-structured database design.

The relative location of an object is contained in six independent scalar values. These values represent the six-DOF of an object floating in space. Audrey models relative location along the x-, y-, and z-axes, as well as rotation around these axes, yaw, pitch, and roll. Other representations of the six-DOF are possible. These six values are also maintained in a homogeneous transform, the *cf-from-parent* slot, for ease of computation. Several standard texts are available for further information on homogeneous transforms.

Audrey also maintains the position of each object in reference to the world-base, the universal coordinate system. The *cf-from-world-base* for each object is also a homogeneous transform, and is maintained to simplify computations. Any object's location in the world is computed by multiplying the parent's position in the world and the location of the object relative to its parent.

An object's *appearance*, the location of its vertices, is stored in a slot called *va-from-cf* as a collection of points around the object's coordinate frame. The location of the vertices and the connectivity of the vertices, called the *edge-list*, determine the appearance of the object. In order to correctly draw the object in the viewport, we must obtain the location of the vertices in world coordinates, from the object's *va-from-world-base* slot. This is accomplished by a simple matrix multiplication of the object's location in world and the location of the object's vertices with respect to the object's coordinate frame.

When an object is drawn on the graphics screen, its *appearance* is stored so that the user may use it to perform activities by clicking on the object in the graphics window.

The *current-visibility* of an object can be modified under user control, and the *default-visibility* of an object is stored so that the original visibility of the object can be reinstated.

Audrey utilizes multiple bit planes in order to speed up graphic animation. Objects which are always stationary will never be redrawn or updated unless the camera is moved. These objects are drawn to a separate bit plane to maintain their graphic quality. Objects which are moveable, but aren't currently moving, are drawn to a separate bit-plane. Objects which are currently undergoing animation are drawn to a third bit-plane since these objects are often re-drawn.

Objects which are moveable have their motion described in parameterized functions which can be applied to the object. For instance, if an object is to slide along its x-axis, then a function can be applied to it which allows that motion. These parameterized functions are stored in a structure called a *port* which contains other information about the function, such as how to read and how to change the current value of the parameter. A collection of ports can be stored in an object's *port-list*.

4.4.4.2 Devices

A device is a list of object-port pairs which describes a linkage which is to move in a coordinated manner. A Unimation PUMA 560 manipulator is a device which consists of a variety of manipulator links and the movement methods which describe the articulation of the manipulator. In the case of a robotic manipulator, there are several methods by which the device can be manipulated. The manipulator may be controlled through its joints or through the end-effector. End-effector motion can occur with respect to a variety of coordinate frames such as world, tool, end-effector, or a defined task frame.

4.4.5 Communication Links

A large part of Audrey's role in TIPS is that of coordinator amongst the various computer programs which make up the system. Audrey accepts commands from the task planner, simulates actions for the user, and queries other computers and computer programs for information.

4.4.5.1 Task Planner Interface

Audrey accepts requests for simulation from the task planner (RMS), and returns an indication of the success or failure of the requests. The most generic simulation request is for the motion of the manipulators. The task planner stores only the names of points to which it would wish to move; if it is necessary to move toward the vicinity of a screw, then the planner would command a move to a point in space called 'SCREW-VICINITY. Audrey can parse this name, breaking it down into its constituent parts, converting the semantic label to a set of numeric values which describe the manipulator position which satisfies the requirement of being in the vicinity of the screw. The task planner can also send sequences of semantic names which represent a path through space which takes the manipulator to a required end-point. Each time a motion is requested by the task planner, Audrey analyzes each position of the path to determine the kinematic feasibility of the command. The motion is determined to be kinematically feasible if each point in the path can be traversed without changing the pose of the arm.

A variety of other commands are also passed to Audrey for simulation which involves grasping and moving objects. These commands are simulated by a series of procedures which emulate the function of the RTC subsystem.

4.4.5.2 Spatial Planner Interface

CENTeR is responsible for finding manipulator paths for the gross motions which are passed to Audrey by the task planner. Gross motions are those large motions which take the manipulator from one location to another without performing any other significant purpose. For instance, the task planner will command a move from the manipulator's safe-home position to a location which is in the vicinity of a tool. This gross motion is then followed by a command to grasp the tool. Moving to grasp is not a gross motion.

Audrey converts the task planner's request for manipulator motion into a request to the spatial planner for a path from the current location of the manipulator to those configurations which satisfy the requested end-point of the motion. The spatial planner returns a path, if one is found; otherwise, no path is returned. The returned path is then simulated in the viewport window, and is an indication of the success of the request action. Should the spatial planner fail to find a path, a failure of the requested action is assumed.

When the Spatial Planner is activated, motion commands from the Task Planner for a particular arm are converted by Audrey into a request to the spatial planner for a manipulator trajectory. The Task Planner stores only the names of Cartesian space locations for the end-effector so that it can command motions to the appropriate location for a task. These names are then matched to the eight possible PUMA configurations which satisfy the Cartesian location of the end-effector. The configurations of the arm which violate joint stops and reach limitations are pruned from the goal point set, and the remaining possibilities are sent to the spatial planner as the possible goal positions of the arm. The current joint values are read from the simulator and output to the spatial planner as the start position of the arm. Should the Spatial Planner be unavailable, and should the motion request from the Task Planner necessitate a pose flip, the user is prompted to design the path through the pose flip. The user can replace the Spatial Planner in order to more closely supervise autonomous operation.

4.4.5.3 Run-Time Controller

Since active use of the robotic hardware by TIPS has not yet been possible, Audrey has been used as a simulator of the RTC subsystem. This allows us to simulate some of the expected failures in a real robotics world. Commands such as Get-Tool are provided to TIPS by the RTC subsystem as macros. The RTC subsystem is responsible for expanding a Get-Tool command into its component steps: Compliant Move, Grasp, Compliant Move. Audrey simulates the success of a Get-Tool command by performing a kinematic analysis on its expanded motion sequence. Other failure tests can also be conducted.

4.4.6 Telerobot Testbed Interface

An operator is able to control many aspects of the Telerobot through Audrey's display screen. A variety of menu picks are defined which set variables or command actions in the various subsystems. The operator has a menu command for each of the commands which the task planner is capable of planning; a command dictionary provided by the Run-Time Controller provides the capabilities for both the operator and the task planner. Subsystem menus are also provided to set modes of motion for the teleoperator and set gains on the FRHCs. All such menu commands cannot possibly be defined on an *a priori* basis; Audrey provides an interface by which any subsystem can request that the operator make a menu pick. The pick is then sent back to the requesting subsystem.

4.5 LESSONS LEARNED

4.5.1 Abstraction for Real Time Processing

Both RMS and CENTeR are fast at execution time. This is due to the fact that each utilizes a model in its reasoning domain which is abstracted from a more complete and precise description of the world. CENTeR derives its model of free-space automatically (off-line) from geometric models of the workspace; RMS' models of satellite structure and the robot are coded by hand. Research is proceeding in developing task plans directly from geometric and Computer-Aided Design (CAD) models, but current prototypes are very slow. Such systems would best be used as off-line preprocessors, analogous to CENTeR's graph generator, to build abstracted data structures to drive run-time execution. The use of abstractions is one effective way to improve run-time performance.

4.5.2 Limited-Domain Reasoning

A common problem encountered in applying AI technology is that reasoning engines which worked on over-simplified versions of the problem bog down when realistic complexity is attempted. Such realistically complex systems also become difficult to maintain. TIPS consists of several reasoning engines which work in very limited domains, but which collectively get the job done. This strategy was not explicitly chosen at the outset, but proved to work well.

4.5.3 Errors as Normal, Expected Behavior

One of the most interesting aspects of RMS is that it can recover from errors without doing deep reasoning about what caused the error. Rather than spending a vast amount of time developing more complex modeling, model updating, and reasoning for feed-forward planning, much less time was spent on making the overall system tolerant of errors. The resulting performance was well beyond expectations.

4.5.4 Separation of "What" and "How"

An early RMS design decision, the separation of planning what should be done in the task-space regardless of agent, from planning how a particular robot should do it, appears to be appropriate.

4.5.5 Steady Target

During the past three years, JPL Telerobot Testbed objectives have changed and target demonstrations have been re-defined and re-scheduled several times. Although this is necessary adjustment for integrating a system of emerging research products, such fluctuation can hinder research. The SARG target of planning the servicing of a Solar Maximum-like satellite was maintained throughout the same three-year period and the consistency of vision was rewarded by the success of TIPS.

5 RESEARCH CONTEXT

5.1 CONTINUING RESEARCH

5.1.1 Architecture: Reasoning Engines, External Devices, and Tasking

The control structure for the current version of TIPS is simple but adequate. As more reasoning engines are added, and to accommodate greater data flow from sensors, the TIPS control architecture must be updated (figure 13). Software falls into three categories: external interface support, knowledge coordination, and domain-specific reasoning engines. The knowledge coordinator will manage tasking. Higher speed collision detection than now exists will be performed on a special-purpose machine. New reasoning engines (listed below) resulting from research will be added.

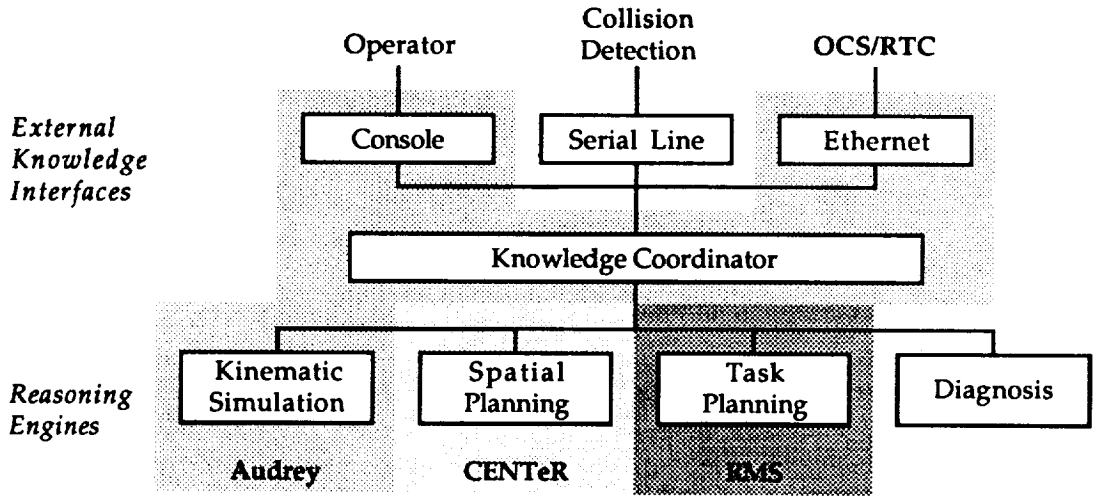


Figure 13. Current TIPS modules and the extended TIPS architecture.

5.1.2 Compliant-Manipulation Planning

The RTC demonstration in FY88 included the opening of a door. Because of kinematic constraints, the entire operation could not be performed from a single grasp. Instead, one arm opened the door part way, and the other arm opened it the rest of the way. This hand-over-hand method of dealing with the door-opening problem was designed by the researchers setting up the demonstration. A robust system would be able to use one arm to open the door automatically.

Indeed, a robust planning system should be able to do anything a human programmer could with the same commands. To this end, a brief investigation was made into extending RMS to be able to do this subdivision of a single operation into component pieces. The dynamic interactive character of RMS seemed suited to this task. It is not clear whether the best approach is to extend RMS to do this, or to develop an independent reasoning engine.

It should be noted that there is some interaction between this problem and that of overall positioning of the telerobot relative to the work-space.

5.1.3 Operator Plan-Editing Interface

The version of TIPS that was demonstrated had the servicing goal hard-wired in. Although the engine was tested with various goals, both single and multiple, the operator interface was not developed.

Stage One of RMS produces a partially-ordered graph of sub-goals. An interactive graphics display which would allow addition, deletion, re-ordering, and forced ordering of initially parallel sub-goals needs to be developed. Some initial design work for such an editor has been done.

An analogous interface to Stage Two of RMS is also needed. This would be an interface to tags on the world knowledge, which would allow the operator to constrain the planner to favor a particular arm or tool, for example.

ART provides output-only displays in its environment, but better interactive capabilities are needed.

5.1.4 Spatial Constraints for Redundant Control

Redundant manipulators have more than 6 joints which determine the 6 parameters of position and orientation of the end-effector. There is an infinite number of inverse kinematic solutions. Control of such manipulators is an active area of research.

Homayoun Seraji of JPL is leading a team investigating the use of spatial constraints imposed by nearby obstacles as additional parameters to complete the specification of manipulator control. Investigations also include the use of inequalities in representing these spatial constraints. Rich Colbaugh and Kristin Glass of New Mexico State University and members of SARG are participating in this research.

This research has some interesting implications for robot control architectures because it links high-level spatial planning directly to low-level servo control.

5.1.5 Space Operations Planning and Telerobotic Task Planning

Before telerobots will be utilized in space, their activities must be integrated into space operations plans. Such planning is driven by resource constraints such as power, crew, and communications links. Recent research has resulted in a planner-scheduler called Plan-IT. Plan-IT has been put to use for Deep Space Network and Spacelab scheduling [G2], and a derivative is under development as part of JPL's multi-mission Space Flight Operations Center.

One possibility under consideration is to replace RMS' Stage One with a planner of this type.

This NASA-specific research into integrating Space Operations and Telerobotic Task Planning must be done to support flight telerobot operations.

5.2 FIELD TRIPS

During FY88, TIPS developers visited several facilities to learn first-hand about teleoperation and satellite servicing. Here are observations from these trips.

5.2.1 JPL Teleoperation Laboratory

The most striking impression from performing teleoperation is the lack of the perception of the violence at the task board, even with force reflection. Although the cameras, force-torque displays, and the force reflection show when resistance is encountered and that the task board is being shaken, the operator does not fully sense the violence. The strongest indication was through vibrations felt through the floor.

With force reflection, one can, with no prior experience, pick up the hand controllers and successfully perform a high-precision insertion task — even without becoming fully acclimated to the live video points of view.

Changing arm poses (such as from elbow down to elbow up) is awkward. It is also not easy to monitor the robot elbow to be sure of avoiding collisions.

5.2.2 The National Bureau of Standards

The goal of the automated factory at NBS is to be able to autonomously and cost-effectively manufacture small batches of parts. This problem differs from NASA's space telerobotics problem in several ways.

At NBS, the part is brought to the tool whereas for satellite servicing, the tool is brought to the part. Much of the telerobot problem is the planning of manipulator motion, application of force, and manipulability. All of this is pre-determined at most of the NBS stations.

The requirements for error detection and recovery are very different. The NBS line (when fully operational) monitors the drift of precision of the machined parts and anticipates when a cutting tool needs to be replaced.

The specifications of the work-space itself are under the control of the NBS line, while the NASA satellite comes as-is and often differs from available CAD models.

The number of sensors integrated into the NBS factory is striking. This suggests that to succeed with autonomous operations, even in significantly controlled environments, many sensors are required.

The NBS vision system anticipates the two-dimensional image and looks for it. This is very different from the JPL system. JPL currently has no path for communication of visual context, which might be known or planned by a planner, to the vision algorithms.

One point in common is that both JPL and NBS seek to complete an overall job rather than isolated specialized tasks. This is in contrast to most robotics research.

5.2.3 Computer Technology Associates

Computer Technology Associates (CTA) is the company that planned the EVA servicing of the Solar Maximum satellite.

The biggest part of the job was coping with the mass of documentation and the various NASA bureaucracies. Extremely detailed plans and procedures had to be worked out in advance. Many teams at different locations had to be coordinated: crew, satellite control, ground EVA simulation, and shuttle operations control.

At the time of performance of the servicing operation, the biggest problem was fitting into the overall schedule, crew time being the tightest constraint. Other driving constraints were thermal, contamination, and power.

Many unanticipated failures occurred during the operation, and re-planning was necessary to recover. CTA had structured their servicing plan into blocks of activity which could be relocated within the overall shuttle operations plan. They stress that the duration of the planned activities must be modeled.

Notice that none of this is yet covered in telerobotics research.

5.2.4 Oak Ridge National Laboratory

Teleoperation is used in hostile environments, currently for undersea oil drilling and the handling of nuclear materials.

The Oak Ridge National Laboratory (ORNL) teleoperator for nuclear fuel reprocessing requires two operators: one controls the two manipulator arms; the other controls the transport, which moves the entire robot, and a hook, which holds the weight of heavy objects. The two operators are positioned where they can see each other; during operation there is a significant amount of communication between them as they coordinate their activities.

The objects being manipulated are all heavy-gauge stainless steel. Nothing is fragile. The operator often will cause the arms to bang into things. A microphone in the workspace provides the best sense of the crashing going on. The operators like force reflection, but it is not yet baseline. Stereo video gives the operators headaches, so they prefer separate views. It takes only two weeks to become proficient with the system.

When asked what autonomous capability would make their work easier, the operators said an automatic tool swap would be very helpful.

The facility is used to test the design both of the manipulators and of nuclear plant hardware. The system is not operational. For this reason, there is no real need for a task planner. The operator interface is simple, providing a complete, though small, set of capabilities.

One characteristic of space teleoperations is that spacecraft are extremely fragile: collisions during teleoperation must absolutely be avoided. For undersea and nuclear applications, collisions occur all the time and are part of normal operations; collisions actually provide spatial feedback to the operator. Only NASA truly needs spatial planning.

6 TIPS AND NASREM

The National Bureau of Standards, together with GSFC, has developed a standard architecture for robotic system design, called NASREM [P5].

6.1 MAPPING OF TIPS ARCHITECTURE INTO NASREM

TIPS is a high-level task planner with execution and operator interfaces that map into several modules within NASREM (figure 14).

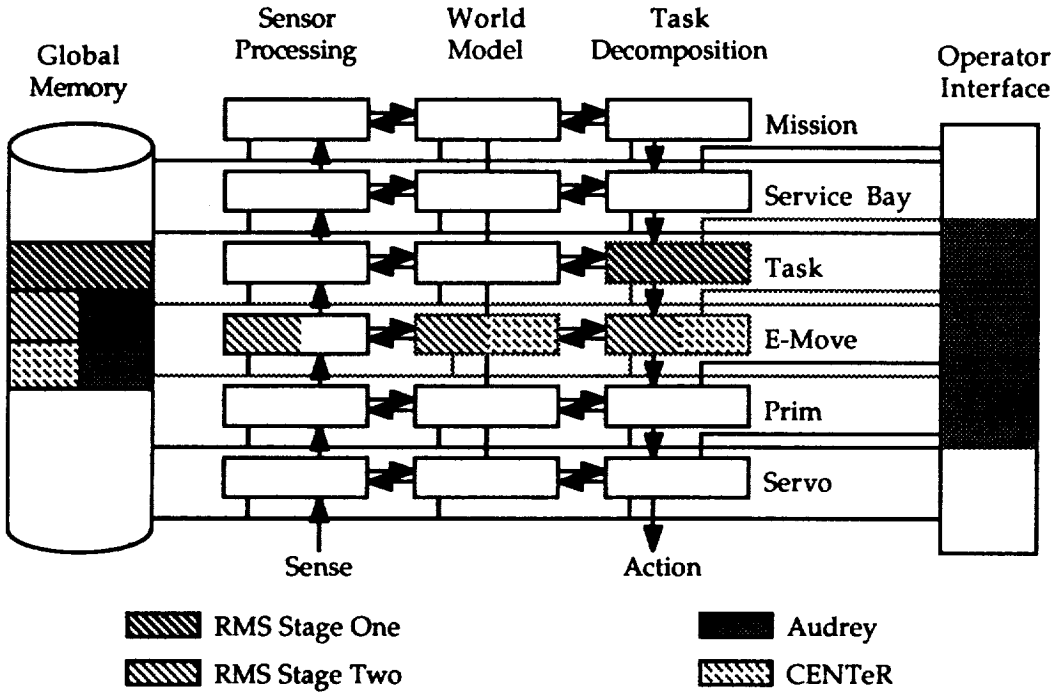


Figure 14. Mapping of TIPS modules into NASREM.

RMS Stage One functionality falls within the NASREM Task level Task Decomposition module. RMS Stage Two functionality falls within NASREM E-Move level Sensor Processing, World Model, and Task Decomposition modules. CENTeR functionality falls within NASREM E-Move level World Model and Task Decomposition modules. The Audrey operator interface functionality falls within the NASREM Operator Interfaces to the Task Decomposition modules at the Task level and the World Model modules at the Task and E-Move levels. The Audrey knowledge coordinator implements the communication between modules. NASREM global data is distributed among the TIPS modules.

6.2 DISCUSSION, DIFFERENCES, AND ISSUES

6.2.1 High-Level Planning

Both TIPS and NASREM separate planning what needs to happen to the object being serviced from planning how the robot should do it: TIPS into the two RMS stages; NASREM into the Task and E-Move levels. This is the clearest parallel between the systems.

6.2.2 Partitioning of Planners and Executors

NASREM divides each Task Decomposition level into three sub-levels: manager, planner, and executor. TIPS is different. Consider RMS Stage Two. It is a single engine, a planner-executor. At the NASREM E-Move level, it receives goals from the higher level and sends commands down the JPL Telerobot Testbed hierarchy to RTC. RTC completes any necessary E-Move level compliant motion planning.

An interesting feature of RMS is that Stage One passes Stage Two an unordered collection of goals to be achieved. Stage Two looks at the current world state and decides which goal to pursue. Insofar as it is choosing which goal it wants to work on, it is a planner; insofar as it works on a goal given to it from a higher-level planner, it is an executor.

6.2.3 Feed-Forward Hierarchy and Feedback Loops

In general, NASREM describes feedback indirectly. Planners within Task Decomposition modules are shown as feed-forward engines; plans are fed forward to executors; and commands are passed down from higher-level modules to lower-level ones. Feedback is carried out through the world, sensor processing, and the world model.

For high-level planners, feedback loops incorporate many modules. It is not clear how effects in the world model are to be correlated with planned actions which caused them.

In the JPL Telerobot Testbed system hierarchy, feedback is hierarchical, like it is in software subroutine calls. Here the mechanism of correlation is clear, but the handling of unexpected events in the world requires bottom-up initiation, in violation of the hierarchy.

Articulation of feedback loops in the system architectural design ensures that both of these issues are addressed.

6.2.4 Partitioning of Space and Time

NASREM provides for parallel planners within a module and describes them as planning activities in different regions of physical space. If these planners are to work together toward solving a single planning problem, coordinating side-effects between the planners could be difficult.

TIPS also has multiple planners working together to build executable plans. These, however, plan for different knowledge domains: RMS handles semantic obstruction; CENTeR handles physical path construction; Audrey handles kinematic feasibility; and RTC handles collision detection. All participate in planning arm motion, yet there is little overlap in the actual parameters they are concerned with.

Also note the differences in the depiction of the architecture containing these multiple planners: NASREM shows a tree while TIPS shows a loop.

6.2.5 Simulation

The handling of simulation in the two systems is somewhat different. The closest thing to simulation in NASREM is the prediction performed by World Model modules. Execution is performed by levels of Task Decomposition modules. In the JPL Telerobot Testbed hierarchy,

the same commands and command path can be used both for simulation and execution. Associated configuration parameters determine where simulators take the place of execution.

TIPS makes extensive use of simulation. The CENTeR map of free-space is constructed by simulating arm positions and resultant collisions. Audrey allows the user to simulate, command the actual robot, or both. Simulation allows the operator to preview a planned action or to verify the correctness of the knowledge base.

RTC also simulates all manipulator motions before executing them.

There is no explicit mention of simulation in NASREM, although simulators will be needed by planners and the operator.

6.2.6 Global Data Base

NASREM specifies a global data base. The passing of messages between directly communicating modules can also be implemented through postings in the global data base. The image is of a single, integrated knowledge base to which all modules have direct access.

TIPS implements a distributed knowledge base. Reasoning engines have direct access only to those parameters they understand. The Audrey knowledge coordinator passes messages among reasoning engines and activates appropriate auxiliary engines to perform needed conversion and completion. When RMS issues a command to move an arm, there is insufficient detail to form a command to RTC, so the Audrey kinematic simulator and CENTeR are each called to fill in missing details. This approach attaches a computing entity to the execution loop, which in turn facilitates incorporation of facilities for monitoring and intervention by the operator or other agent.

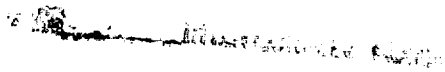
A similar distributed knowledge approach is used throughout the JPL Telerobot Testbed architecture. Overall geometric characteristics of objects are shared by S&P, RTC, and TPR subsystems, but each is interested in details which don't concern the others: S&P computes apparent edges of slightly rounded physical edges, RTC maintains object frames in such a way that simple moves can be represented by pure translations or rotations, and TIPS' Task Planner (RMS) is concerned with semantic relationships among objects. It is not necessary that all parameters be available to all reasoning engines.

There is another consideration which favors the distributed approach. Different parts of the system need to access and update data at different rates. A distributed system allows high-rate, low-volume data and high-volume, low-rate data to be maintained on different media.

When coordination of a distributed database is achieved, the database functions as if it were a single global data base.

ABOUT THE DOCUMENT

The text of this document was created with Microsoft Word version 4.0 on an Apple Macintosh II. The figures were created in MacDraft and MacDraw. The font used was Palatino in the ten and twelve point sizes.



INTENTIONALLY BLANK

TECHNICAL REPORT STANDARD TITLE PAGE

1. Report No. 90-37	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle Planning and Reasoning in the JPL Telerobot Testbed		5. Report Date September 15, 1990 6. Performing Organization Code
7. Author(s) Stephen Peters, David Mittman, Carol Collins, Jacquie O'Meara, Mark Rokey		8. Performing Organization Report No.
9. Performing Organization Name and Address JET PROPULSION LABORATORY California Institute of Technology 4800 Oak Grove Drive Pasadena, California 91109		10. Work Unit No. 11. Contract or Grant No. NAS7-918 13. Type of Report and Period Covered JPL Publication
12. Sponsoring Agency Name and Address NATIONAL AERONAUTICS AND SPACE ADMINISTRATION Washington, D.C. 20546		14. Sponsoring Agency Code B-055-00-00-00-00

15. Supplementary Notes

16. Abstract

The Telerobot Interactive Planning System is being developed at the Jet Propulsion Laboratory to serve as the highest autonomous-control level of the Telerobot Testbed. This publication describes a recent prototype which integrates an operator interface for supervisory control, a task planner supporting disassembly and re-assembly operations, and a spatial planner for collision-free manipulator motion through the workspace. Each of these components is described in detail. Descriptions of the technical problem, approach, and lessons learned are included.

17. Key Words (Selected by Author(s))	18. Distribution Statement
Astronautics (General) Cybernetics	Unclassified; unlimited

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price
Unclassified	Unclassified	73	